W4153 – Cloud Computing

Lecture 6:

REST(4), OAuth2, Middleware, Composition

COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

Contents

Contents

- Questions?
- Discuss Sprint 2

Sprint 2 – Discussion

Sprint 2 – Objectives

- For all microservices:
  - OpenAPI documentation.
  - Good response codes and error codes.
  - Links for HATEOS

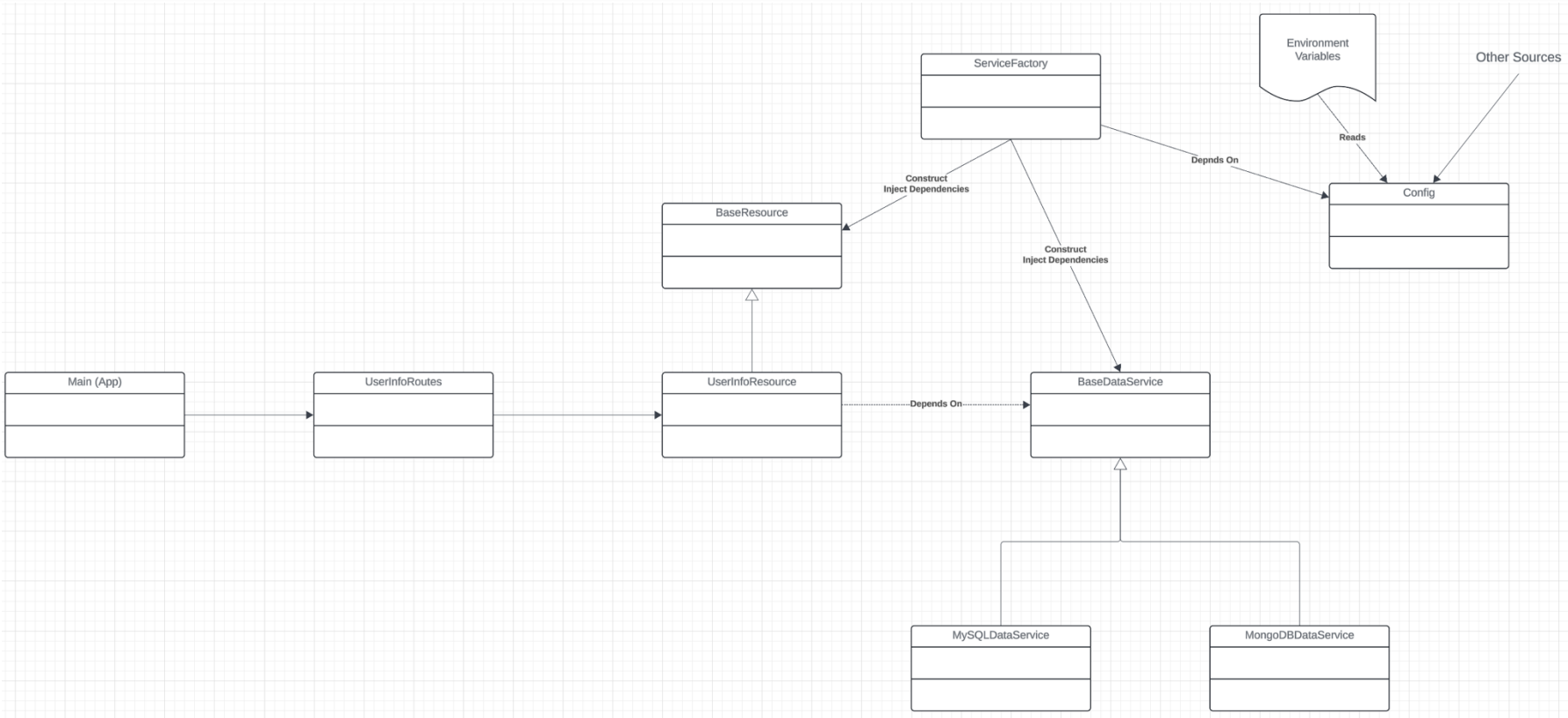
- At least one microservice implementing a complete REST interface and accessing a database. Complete means:
  - All methods and on all paths work/Have basic functionality for CRUD working.
    - /api/courses: GET, POST
    - /api/courses/{call_no}: GET, PUT, DELETE
  - On collection resources, e.g. /api/courses
    - Query string with parameters.
    - Pagination.
    - 201 Created with a link header for a POST. Implement GET on link header for created resource.

- Simple web UI that calls some of the services.

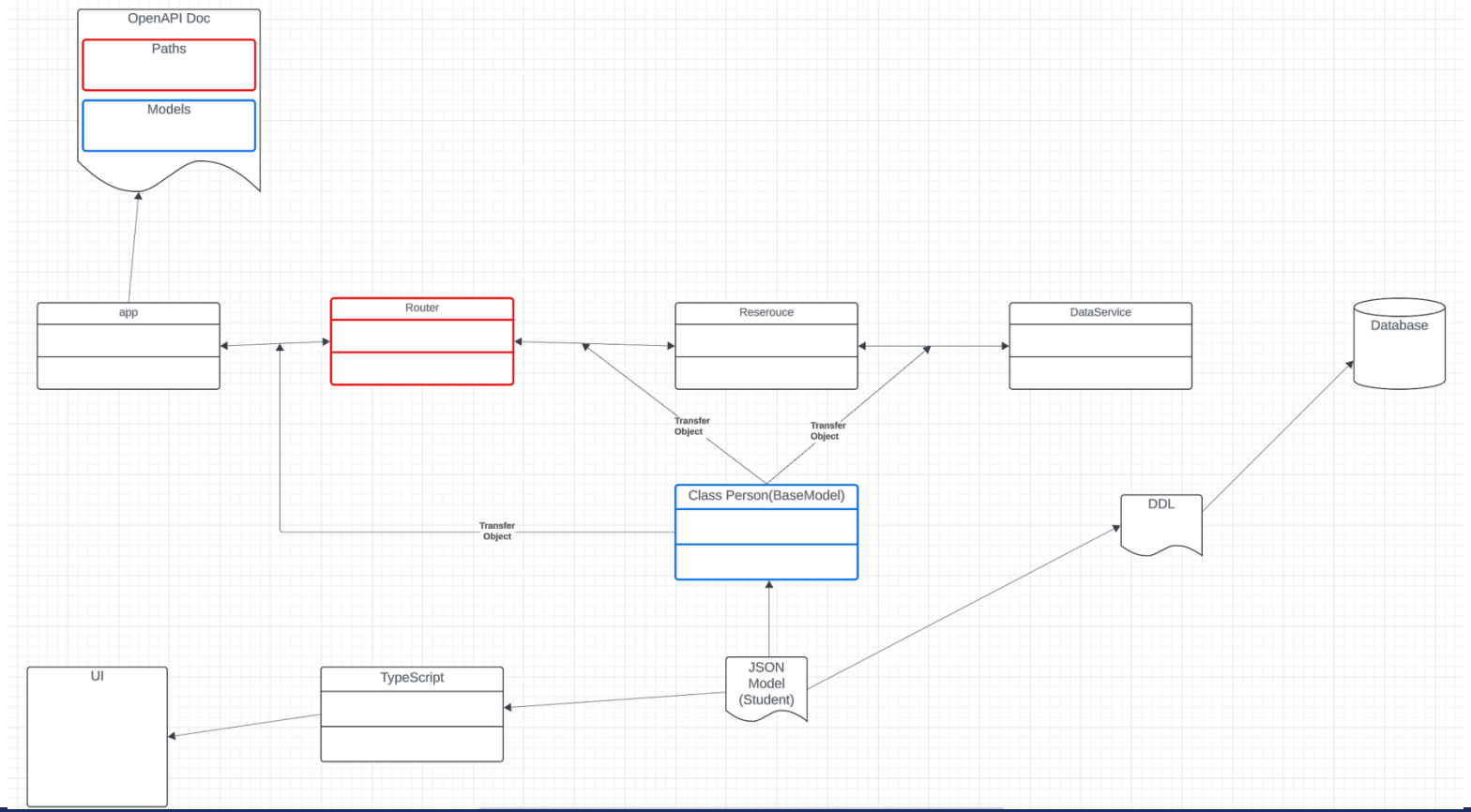
Sprint 2 – Objectives

- One composite microservice/resource, e.g. /api/student_infos, /api/student_infos/{uni}:
  - Calls/delegates to atomic services for GET and POST.
  - 202 Accepted and implementing an asynchronous update to a URL.
  - Implement one method synchronously and one asynchronously using threads, co-routines, …
  - Basic support for navigation paths, including query parameters, e.g. /student_infos/{uni}/courses, /student_infos/{uni}/teams
  - Middleware on each resource that implements before and after logging for all methods/paths.
  - Getting the URLs/dependencies from the environment variables.
- Deploy the composite and components and test.
- OAuth2, OpenIDC using Google login for composite service.
- Overall project structure in GitHub:
  - Linked repositories.
  - First pass at a Trello board with a backlog.

Framework



Framework



Project Structure

https://medium.com/@amirm.lavasani/how-to-structure-your-fastapi-projects-0219a6600a8f

Key Principles for Structuring Projects

In building FastAPI applications, it's crucial to follow these best practices:

1. **Separation of Concerns:** *Separate different aspects of your FastAPI project, such as routes, models, and business logic, for clarity and maintainability.*
2. **Modularization:** *Break down your FastAPI application into reusable modules to promote code reusability and organization.*
3. **Dependency Injection:** *Decouple components by implementing dependency injection, which allows for more flexible and testable code in FastAPI using libraries like* `fastapi.Dependencies`.
4. **Testability:** *Prioritize writing testable code in FastAPI applications by employing strategies like dependency injection and mocking to ensure robust testing and quality assurance.*

Project Structure

Based on File Type

```
.
├── app  # Contains the main application files.
│   ├── __init__.py   # this file makes "app" a "Python package"
│   ├── main.py       # Initializes the FastAPI application.
│   ├── dependencies.py # Defines dependencies used by the routers
│   ├── routers
│   │   ├── __init__.py
│   │   ├── items.py  # Defines routes and endpoints related to items.
│   │   └── users.py  # Defines routes and endpoints related to users.
│   ├── crud
│   │   ├── __init__.py
│   │   ├── item.py  # Defines CRUD operations for items.
│   │   └── user.py  # Defines CRUD operations for users.
│   ├── schemas
│   │   ├── __init__.py
│   │   ├── item.py  # Defines schemas for items.
│   │   └── user.py  # Defines schemas for users.
│   ├── models
│   │   ├── __init__.py
│   │   ├── item.py  # Defines database models for items.
│   │   └── user.py  # Defines database models for users.
│   ├── external_services
│   │   ├── __init__.py
│   │   ├── email.py          # Defines functions for sending emails.
│   │   └── notification.py   # Defines functions for sending notifications
│   └── utils
│       ├── __init__.py
│       ├── authentication.py  # Defines functions for authentication.
│       └── validation.py      # Defines functions for validation.
├── tests
│   ├── __init__.py
│   ├── test_main.py
│   ├── test_items.py  # Tests for the items module.
│   └── test_users.py  # Tests for the users module.
├── requirements.txt
├── .gitignore
└── README.md
```

Module Functionality

```
├── src
│   ├── auth
│   │   ├── router.py         # auth main router with all the endpoints
│   │   ├── schemas.py        # pydantic models
│   │   ├── models.py         # database models
│   │   ├── dependencies.py   # router dependencies
│   │   ├── config.py         # local configs
│   │   ├── constants.py      # module-specific constants
│   │   ├── exceptions.py     # module-specific errors
│   │   ├── service.py        # module-specific business logic
│   │   └── utils.py          # any other non-business logic functions
│   ├── aws
│   │   ├── client.py  # client model for external service communication
│   │   ├── schemas.py
│   │   ├── config.py
│   │   ├── constants.py
│   │   ├── exceptions.py
│   │   └── utils.py
│   └── posts
│   │   ├── router.py
│   │   ├── schemas.py
│   │   ├── models.py
│   │   ├── dependencies.py
│   │   ├── constants.py
│   │   ├── exceptions.py
│   │   ├── service.py
│   │   └── utils.py
│   ├── config.py      # global configs
│   ├── models.py      # global database models
│   ├── exceptions.py  # global exceptions
│   ├── pagination.py  # global module e.g. pagination
│   ├── database.py    # db connection related stuff
│   └── main.py
├── tests/
│   ├── auth
│   ├── aws
│   └── posts
├── templates/
│   └── index.html
├── requirements
│   ├── base.txt
│   ├── dev.txt
│   └── prod.txt
├── .env
├── .gitignore
├── logging.ini
└── alembic.ini
```

Example

- https://github.com/zhanymkanov/fastapi-best-practices

OpenID, OAuth2

OAuth2, OpenIDC

OpenIDC and Oauth2

- **OAuth 2.0**
  - OAuth 2.0 is a protocol for **authorizing client applications** to access protected resources. OAuth 2.0 provides consented access and restricts actions of what the client application can perform on resources on behalf of the user, without ever sharing the user's credentials.
- **OpenID Connect**
  - OpenID Connect is an identity layer on top of the OAuth 2.0 protocol. It allows Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User in an interoperable manner.
  - OpenID exists to enable **Federated Authentication**, where users are able to authenticate with the same identity across multiple web applications. Both users and web applications trust identity providers, such as Google, Yahoo!, and Facebook, to store user profile information and authenticate users on behalf of the application. This eliminates the need for each web application to build its own custom authentication system, and it makes it much easier and faster for users to sign up and sign into sites around the Web.

https://medium.com/software-development-turkey/understanding-oauth-2-0-and-openid-connect-777eb1fc27f

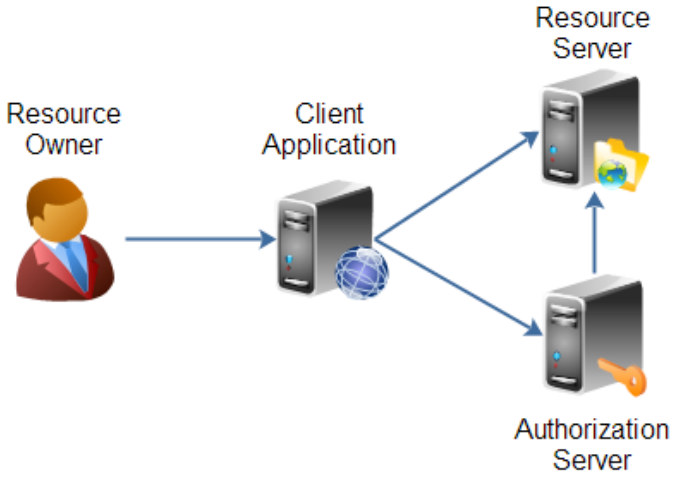
Overview (http://tutorials.jenkov.com/oauth2/index.html)

- Resource Owner
  - Controls *access* to the "data."
  - Facebook **user**, LinkedIn **user**, …
- Resource Server
  - The website that holds/manages info.
  - Facebook, LinkedIn, …
  - And provides access API.
- Client Application
  - "The product you implemented."
  - Wants to read/update
    - "On your behalf"
    - The data the data that the Resource Server maintains
- Authorization Server
  - Grants/rejects authorization
  - Based on Resource Owner decisions.
  - Usually (logically) the same as Resource Server.

OAuth 2.0 defines the following roles of users and applications:

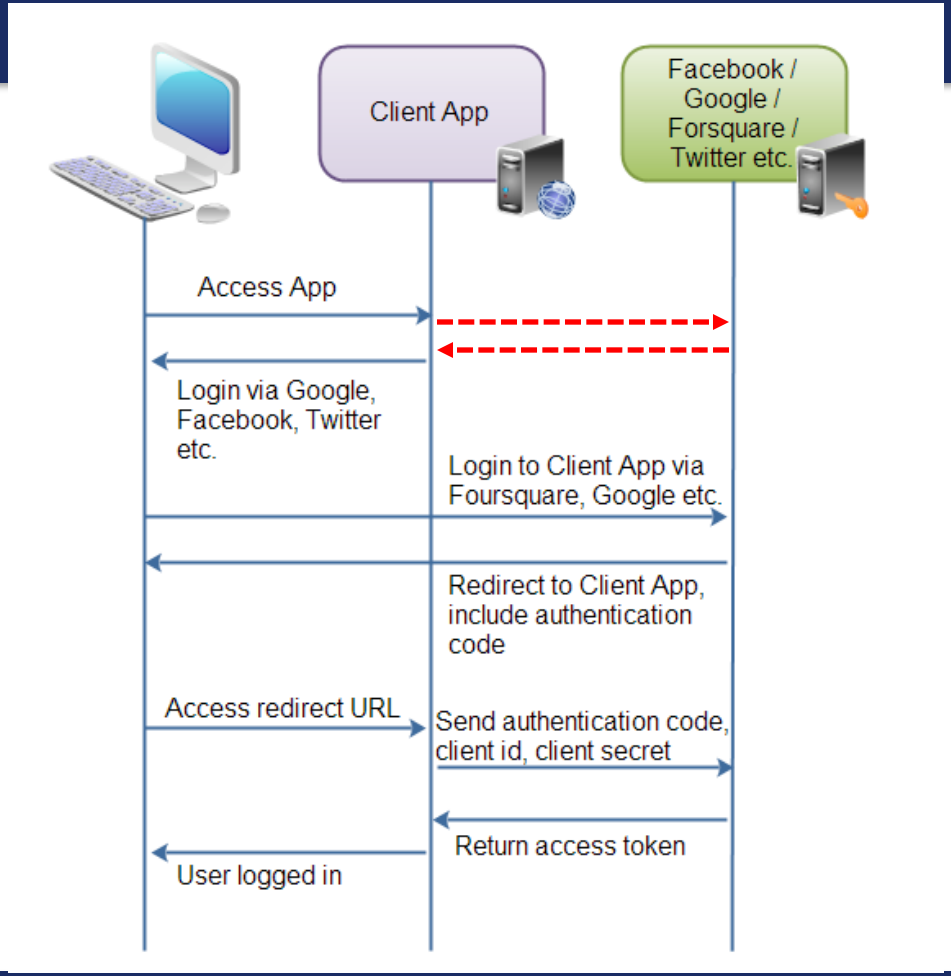
- Resource Owner
- Resource Server
- Client Application
- Authorization Server

These roles are illustrated in this diagram:

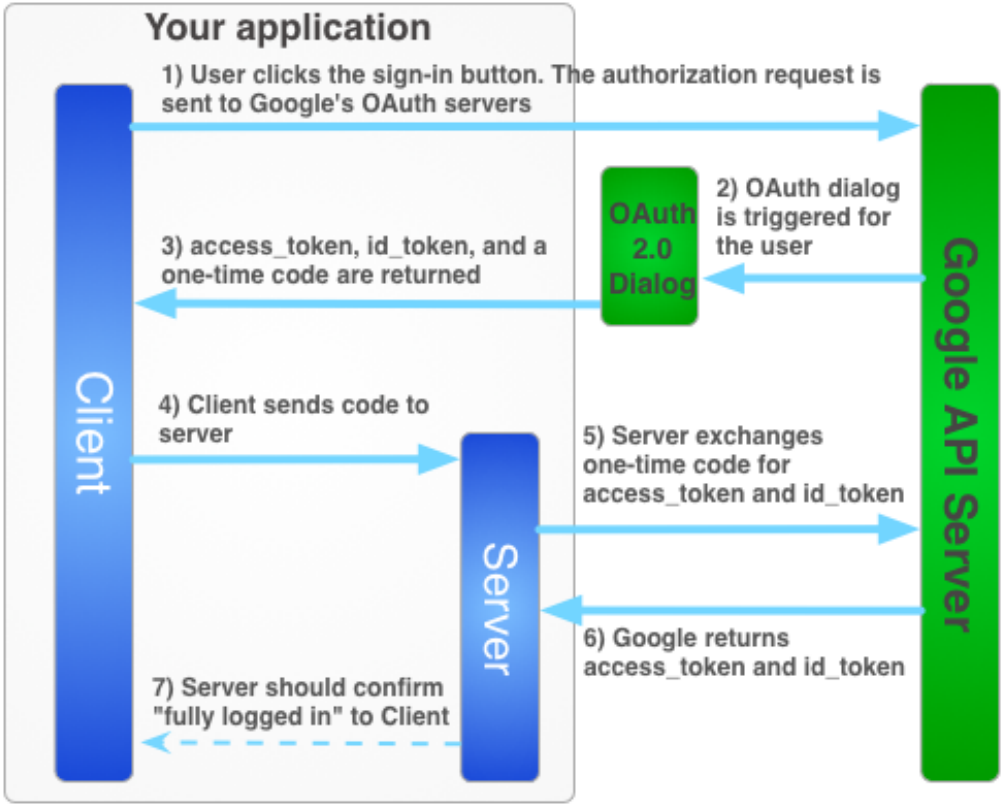


Roles/Flows

- User Clicks “Logon with XXX”
  - Redirect user to XXX
  - With Client App application ID.
  - And permissions app requests.
  - Code MAY have to call Resource Server to get a token to include in redirect URL.
- Browser redirected to XXX
  - Logon on to XXX prompt.
  - Followed by “do you grant permission to …”
  - Clicking button drives a redirect back to Client App. URL contains a temporary token.
- User/Browser
  - Redirected to Client App URL with token.
  - Client App calls XXX API
  - Obtains access token.
  - Returns to User.
- Client App can use access token on API calls.



Google



Walkthrough

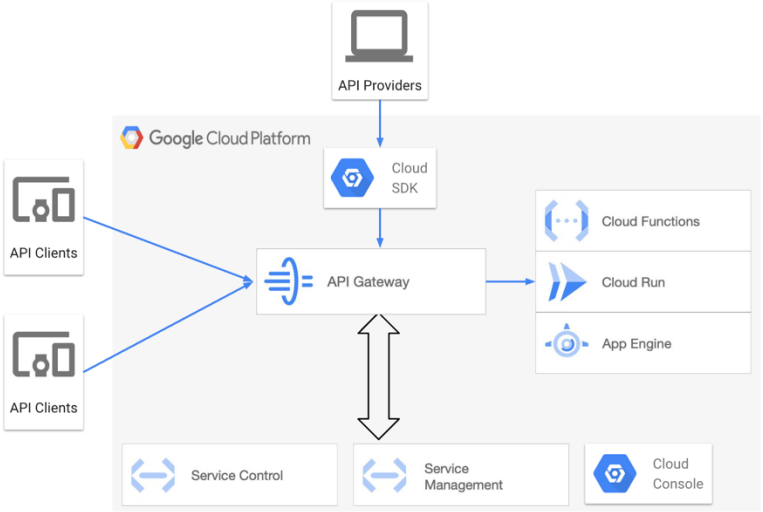
- There are good tutorials: https://medium.com/software-development-turkey/understanding-oauth-2-0-and-openid-connect-777eb1fc27f

- Show Coupon App code:
  - /Users/donald.ferguson/Dropbox/000/00-Current-Repos/student-course-application/course-coupons
  - dff_framework: File system link but would be pip install.
    - Environment variables.
    - Base classes and abstractions
    - Service factory
    - Config and basic dependency injection

- Google Cloud setup
  - Consent
  - URLs

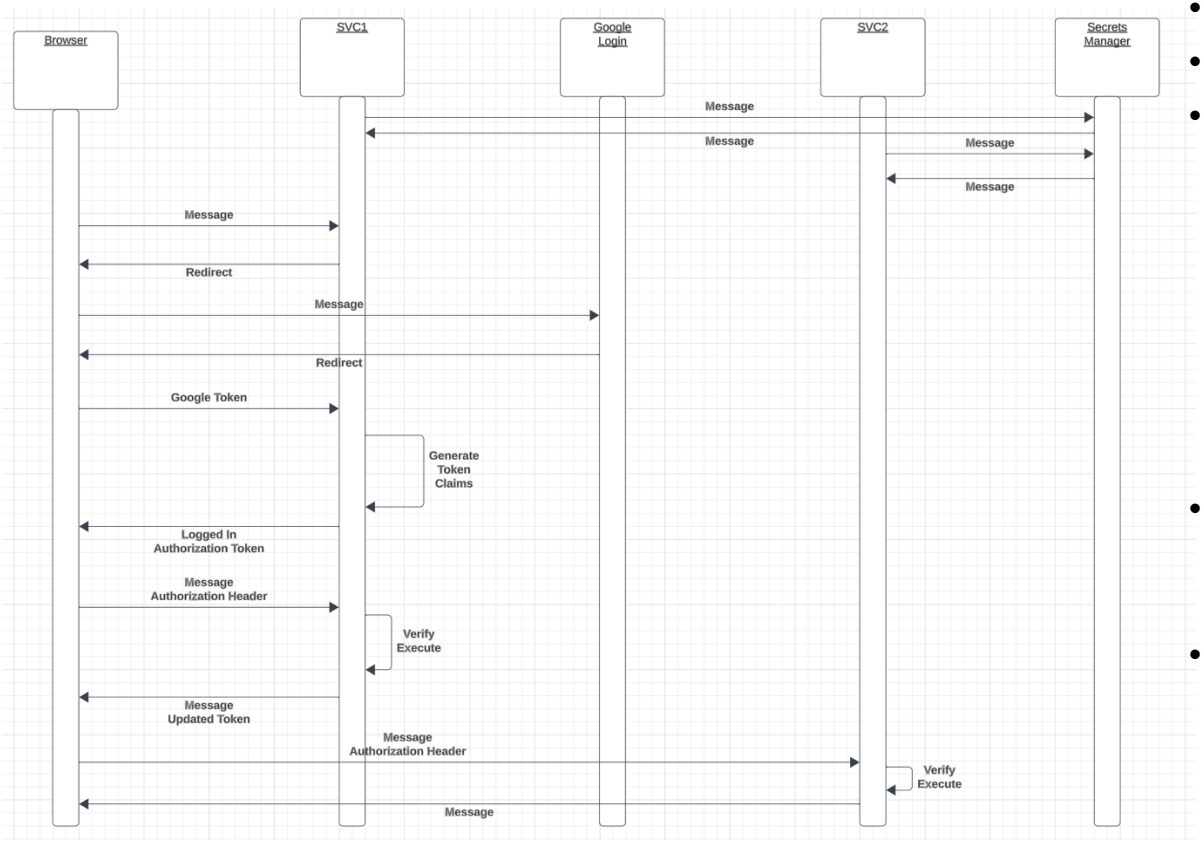
JWT

API Gateway



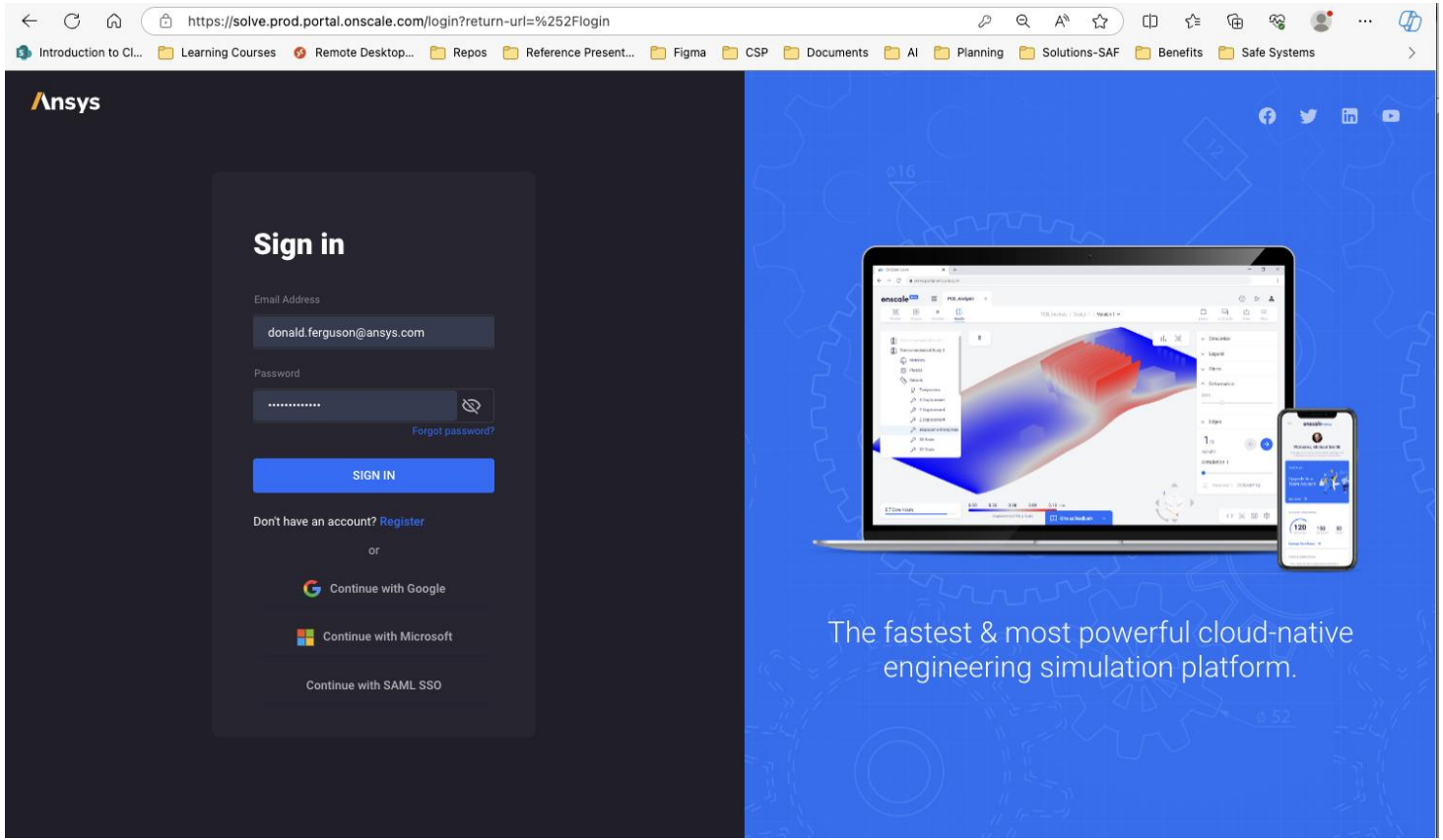
- API Management and API Gateways are fundamental concepts.
- Vastly simplifies changing microservice architectures, monitoring, enforcing security, throttling, versioning, … …
- We will do some simple stuff with a gateway, starting with security.

Login and Authorization Tokens

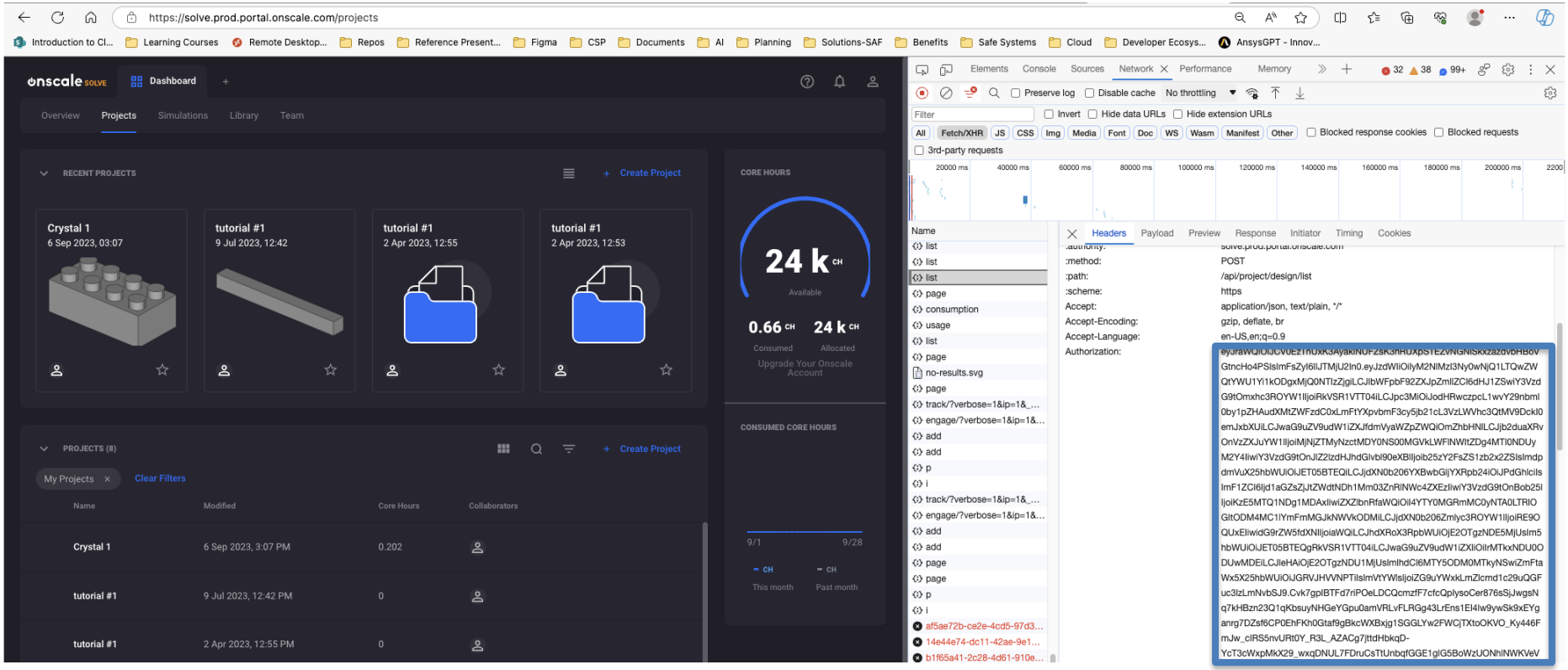


- We have seen Google Login.
- Forms the basics of authentication.
- The server applications:
  - ”Look up” user info in their database and systems.
  - Determine what operations the user can perform.
  - Generate a JWT Authorization containing signed claims.
  - Returns the token in an authorization header.
- The client
  - “Saves” the token.
  - Attached to subsequent calls.
- Servers can then authorize based on
  - Verifying the signature.
  - Checking that the client has a claim authorizing the operation.

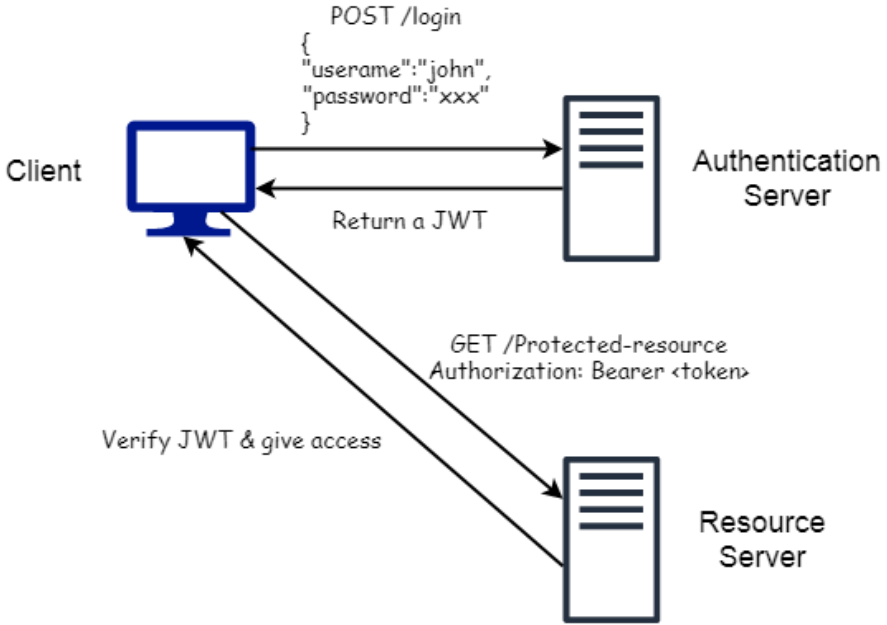
Real Example



Real Example



JWT



When should you use JSON Web Tokens?

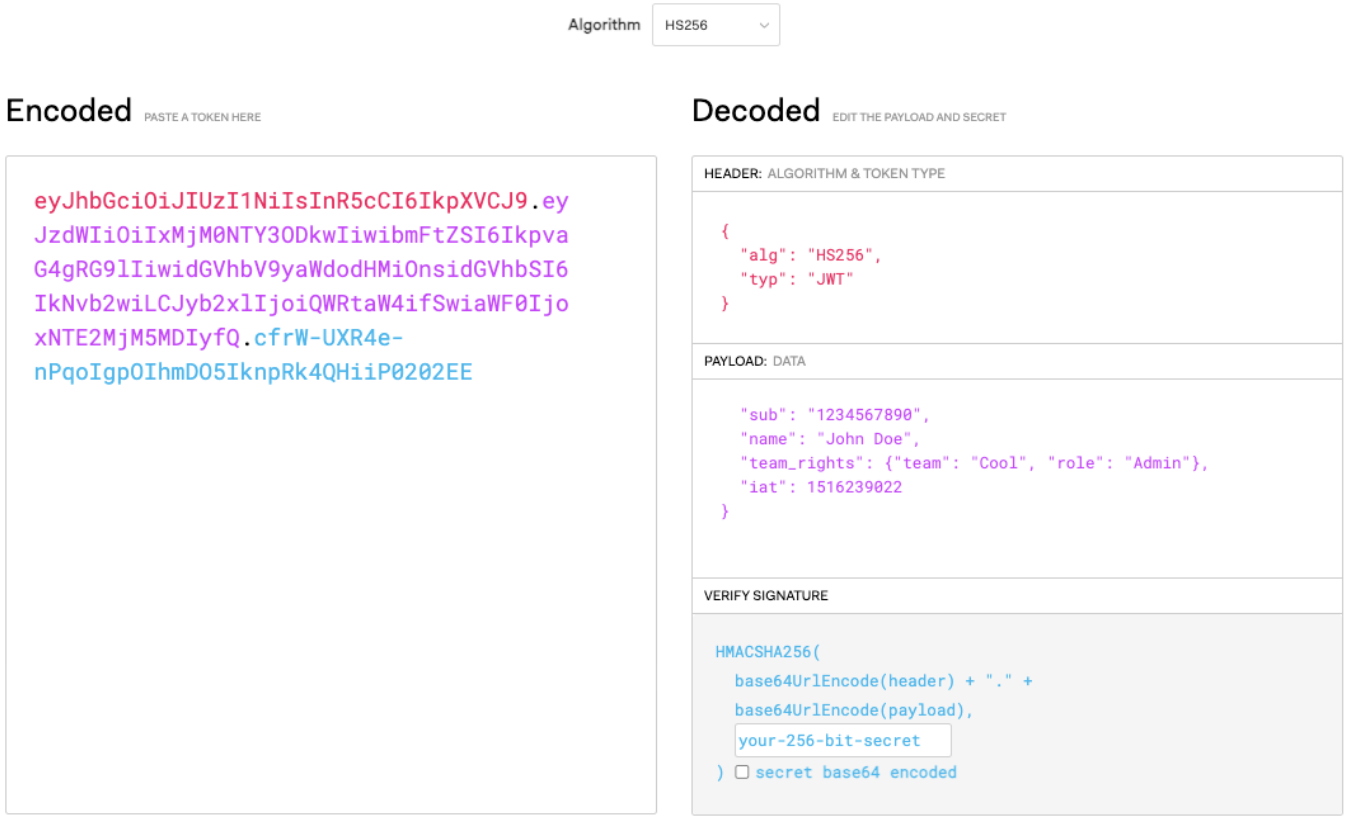
Here are some scenarios where JSON Web Tokens are useful:

- **Authorization**: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.

- **Information Exchange**: JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

https://medium.com/swlh/all-you-need-to-know-about-json-web-token-jwt-8a5d6131157f

JWT.io



REST Continued

Pagination, Hyper Linked

- Bad things may happen when a GET returns a huge result set.
- A *page* is a window on the large result set.
- *limit* is the no. of entries in the result set.
  - Client may specific a limit.
  - Server may have a default and/or maximum limit it allows.
- There are many design patterns for implementing pagination.
- Simplest is mirroring the SQL model:
  - *limit=xxx&offset=yyy*
  - The links section is the response should contain:
    - prev
    - next
    - current
    - May contain *first* and *last.*

Query Parameters

2.5. Use query component to filter URI collection

Often, you will encounter requirements where you will need a collection of resources sorted, filtered, or limited based on some specific resource attribute.

For this requirement, do not create new APIs – instead, enable sorting, filtering, and pagination capabilities in resource collection API and pass the input parameters as query parameters. e.g.

```
/device-management/managed-devices
/device-management/managed-devices?region=USA
/device-management/managed-devices?region=USA&brand=XYZ
/device-management/managed-devices?region=USA&brand=XYZ&sort=installation-date
```

- The basics of query string is very common, e.g. ?region=USA&brand=XYZ
- Sort/Order by is not “standard” and APIs pick and document a convention.
- Project (choosing a subset of properties) is also not standard. APIs picks and document a convention.

POST -- Creating a Resource

POST methods

If a POST method creates a new resource, it returns HTTP status code 201 (Created). The URI of the new resource is included in the Location header of the response. The response body contains a representation of the resource.

If the method does some processing but does not create a new resource, the method can return HTTP status code 200 and include the result of the operation in the response body. Alternatively, if there is no result to return, the method can return HTTP status code 204 (No Content) with no response body.

If the client puts invalid data into the request, the server should return HTTP status code 400 (Bad Request). The response body can contain additional information about the error or a link to a URI that provides more details.

- POST sends a body to “create a new resource.”
- The identity/URL of the new resource is not always obvious.
- POST returns 201 Created with a link header containing the URL to the newly created resource instance.

POST

Receiving a response indicating user creation

Let's assume there's a REST API for managing users with an endpoint at `http://example.com/users`.
In this example, we send a `POST` request with the following body to create a user:

HTTP

```
POST /users HTTP/1.1
Host: example.com
Content-Type: application/json

{
  "firstName": "Brian",
  "lastName": "Smith",
  "email": "brian.smith@example.com"
}
```

After successful user creation, the `201 Created` response will look like this:

HTTP

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: http://example.com/users/123

{
  "message": "New user created",
  "user": {
    "id": 123,
    "firstName": "Brian",
    "lastName": "Smith",
    "email": "brian.smith@example.com"
  }
}
```

202 Accepted

HTTP Status Code 202: Accepted

The HTTP status code `202` is used to indicate that a request has been accepted for processing, but the processing has not yet been completed. This code is often used when a server is processing a request asynchronously and does not have an immediate response.

For example, if a client sends a request to a server to perform a long-running task, such as generating a report, the server might respond with a status code of `202` to indicate that the request has been accepted and is being processed. The client can then check back later to see the status of the task.

The Location header is used to provide a URL that the client can use to check the status of the asynchronous task that was requested. The value of the Location header (`/reports/123`) is simply a unique identifier for the task and does not necessarily have to correspond to a specific resource on the server.

The Location header is simply a way for the server to provide the client with a way to track the progress of the task. The client can send a request to the URL provided in the Location header to check the status of the task and get any additional information that may be available.

Request

```
POST /reports/generate HTTP/1.1
Host: example.com
Content-Type: application/json

{
  "start_date": "2022-01-01",
  "end_date": "2022-01-31"
}
```

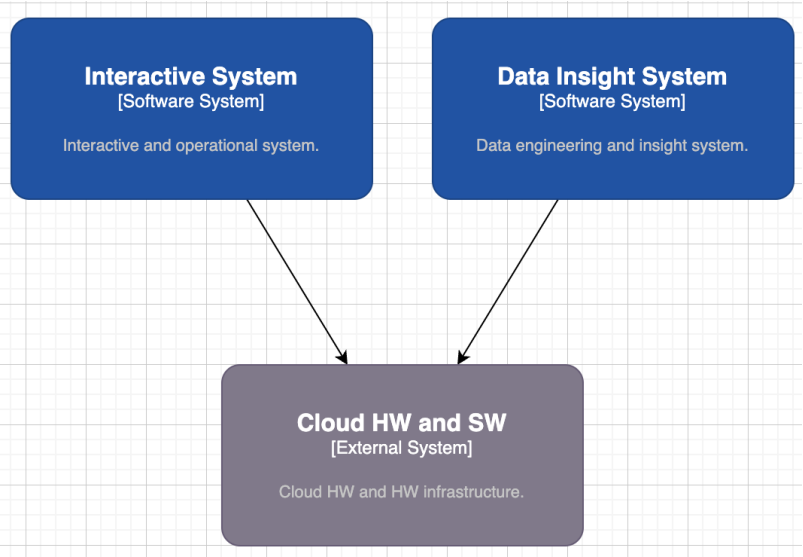
Response

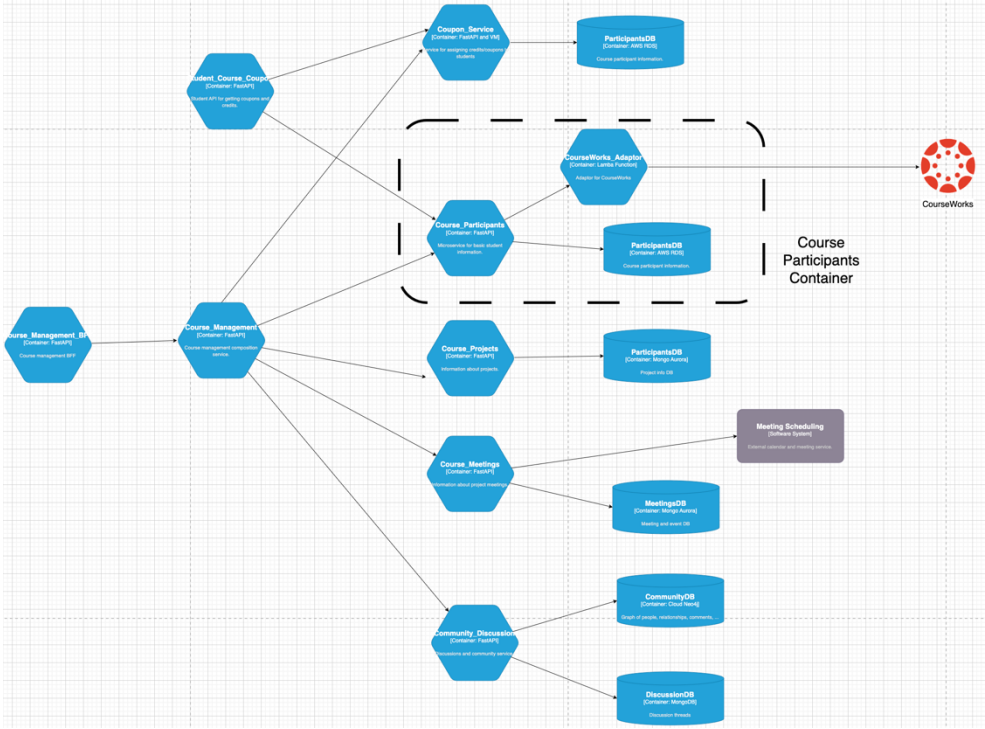
```
HTTP/1.1 202 Accepted
Content-Type: application/json
Location: /reports/123

{
  "id": 123,
  "status": "processing"
}
```

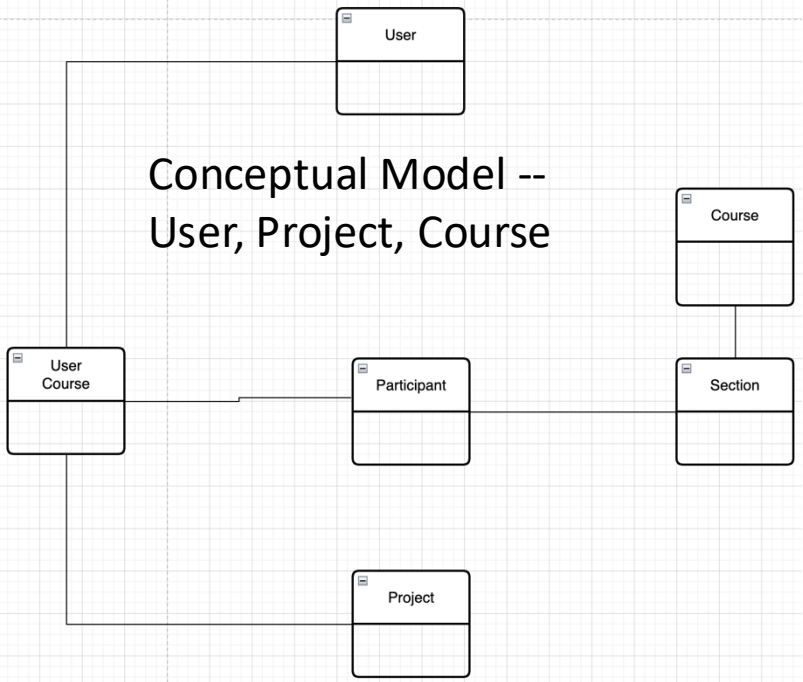
Composition

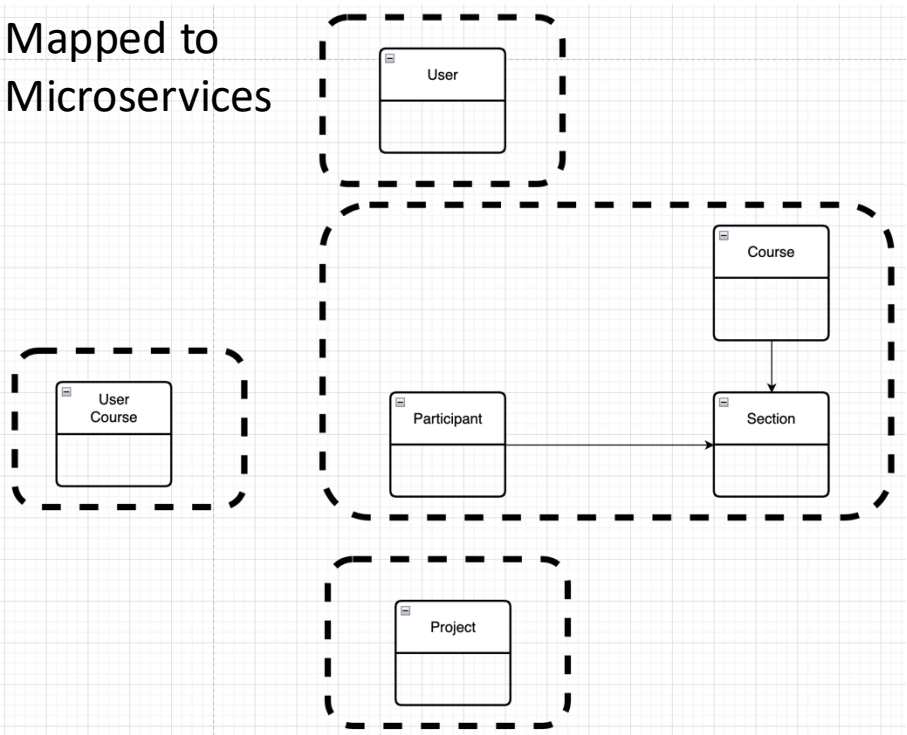
(Partial) Logical View – Interactive System





Resource Oriented and Microservices – My Project



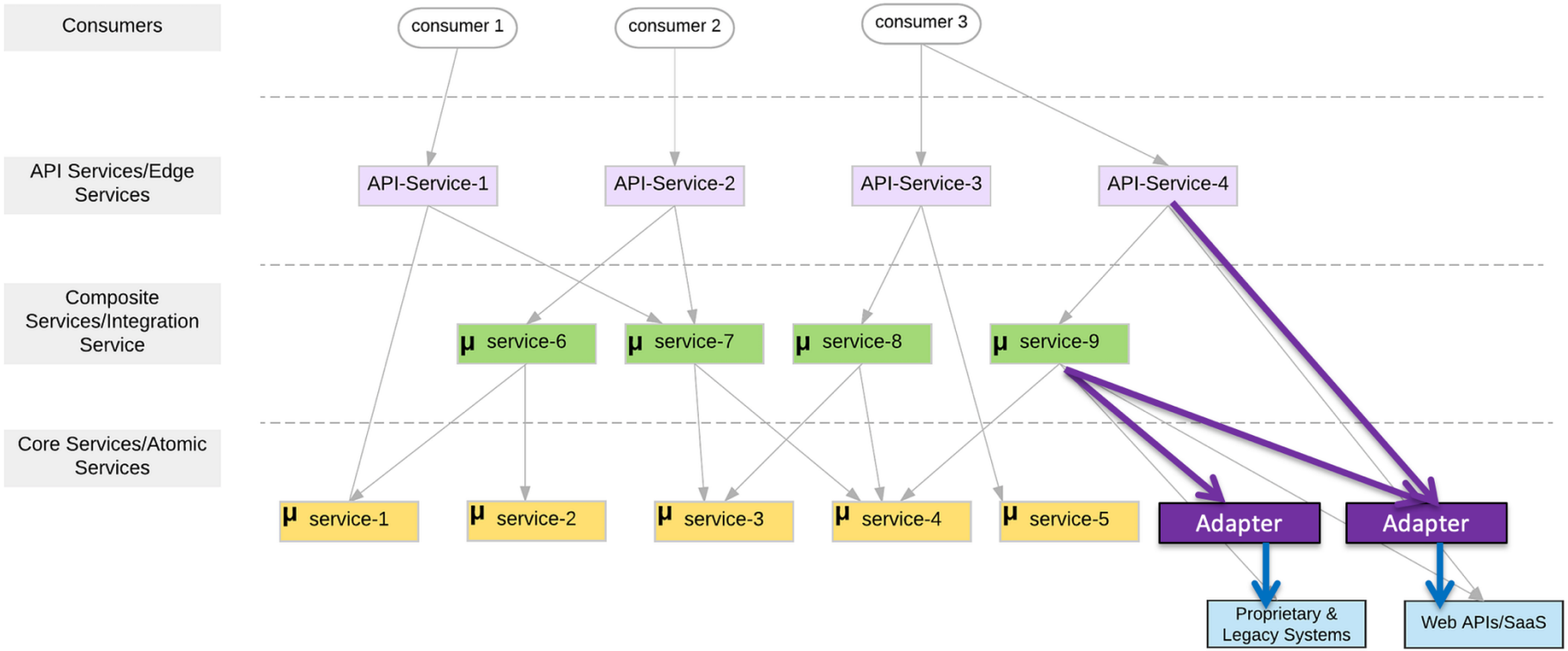


This creates some interesting challenges, e.g.

- Joins
- Foreign Keys

If the cluster of resources might be useful in other solutions → It may be a microservice.

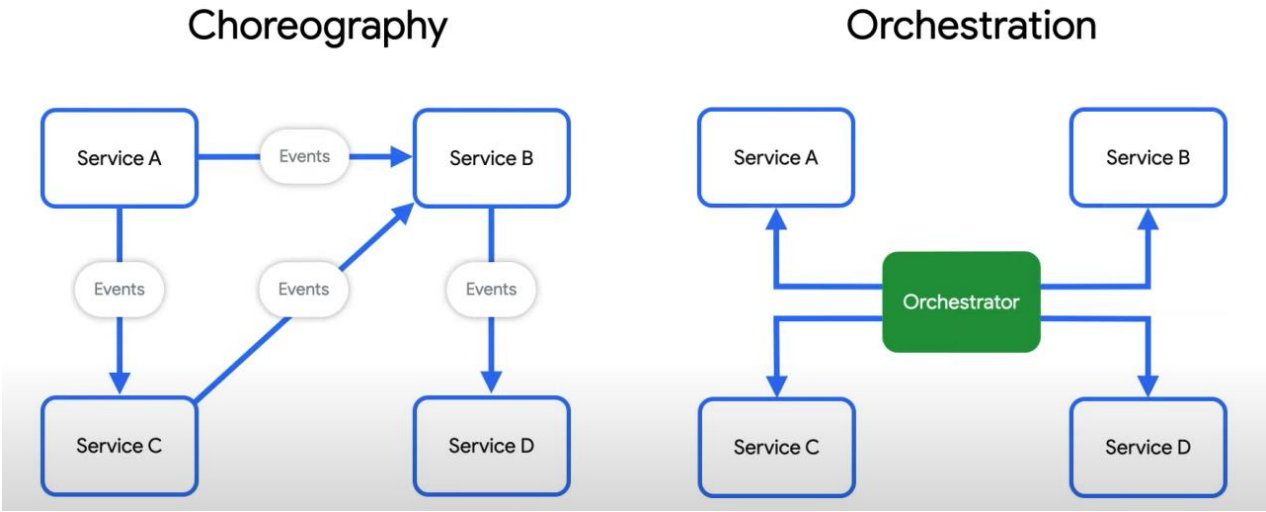
Atomic and Composition



First Group of Microservices

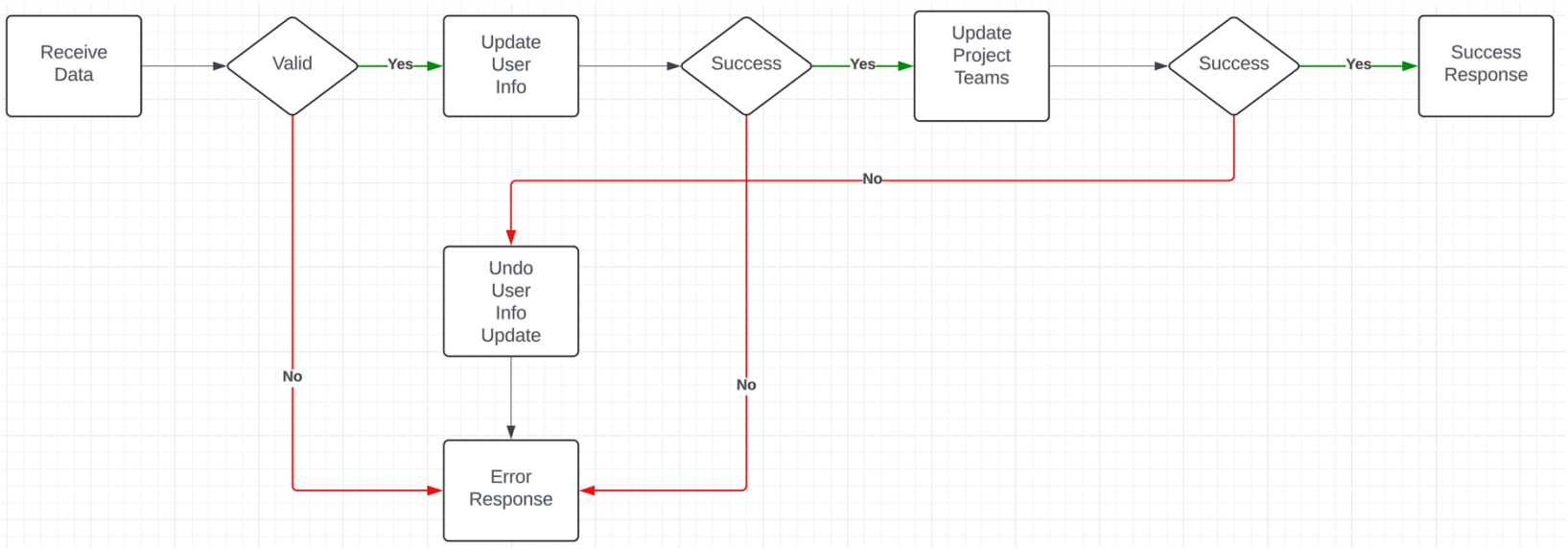
- My first sprint with have 5 microservices:
  - 5 core/atomic microservices:
    - user_info
    - courseworks_adaptor
    - course_info
    - student_coupons
    - project_teams
  - 1 composite/integration microservice: core_course_management
- The composite service:
  - Implements “consistency” in the data in the core services, e.g. “foreign keys.”
  - Eliminates the need for complex logic in the UI or client, which would be impacted by changes.
  - Exposes only information from the basic services that the client requires.
- The composite service is conceptually similar to a database view.

Behavioral Composition



- There are two base patterns: choreography and orchestration.
- There also several different implementation technologies and choices.
- And many related concepts, e.g. Sagas, Event Driven Architectures, Pub/Sub, … …
- We will cover and explore incrementally, but for now we will keep it simple and do traditional “if … then …” code.

Composite Microservice Logic



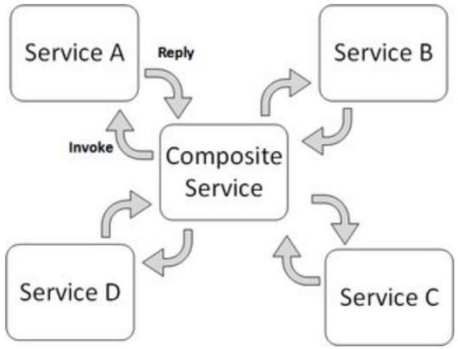
- The GET on the composite invokes GET on several atomic services. This can occur synchronously or asynchronously in parallel.
- PUT, POST and DELETE are more complex. Your logic may have to undo some delegated updates that occur if others fail.
- To get started, we will just use simple logic but explore more complex approaches.

Concepts

Service orchestration

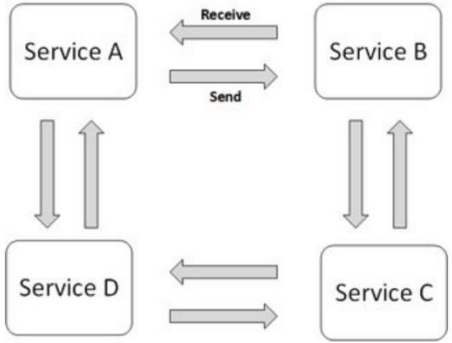
Service orchestration represents a single centralized executable business process (the orchestrator) that coordinates the interaction among different services. The orchestrator is responsible for invoking and combining the services.

The relationship between all the participating services are described by a single endpoint (i.e the composite service). The orchestration includes the management of transactions between individual services. Orchestration employs a centralized approach for service composition.



Service Choreography

Service choreography is a global description of the participating services, which is defined by exchange of messages, rules of interaction and agreements between two or more endpoints. Choreography employs a decentralized approach for service composition.

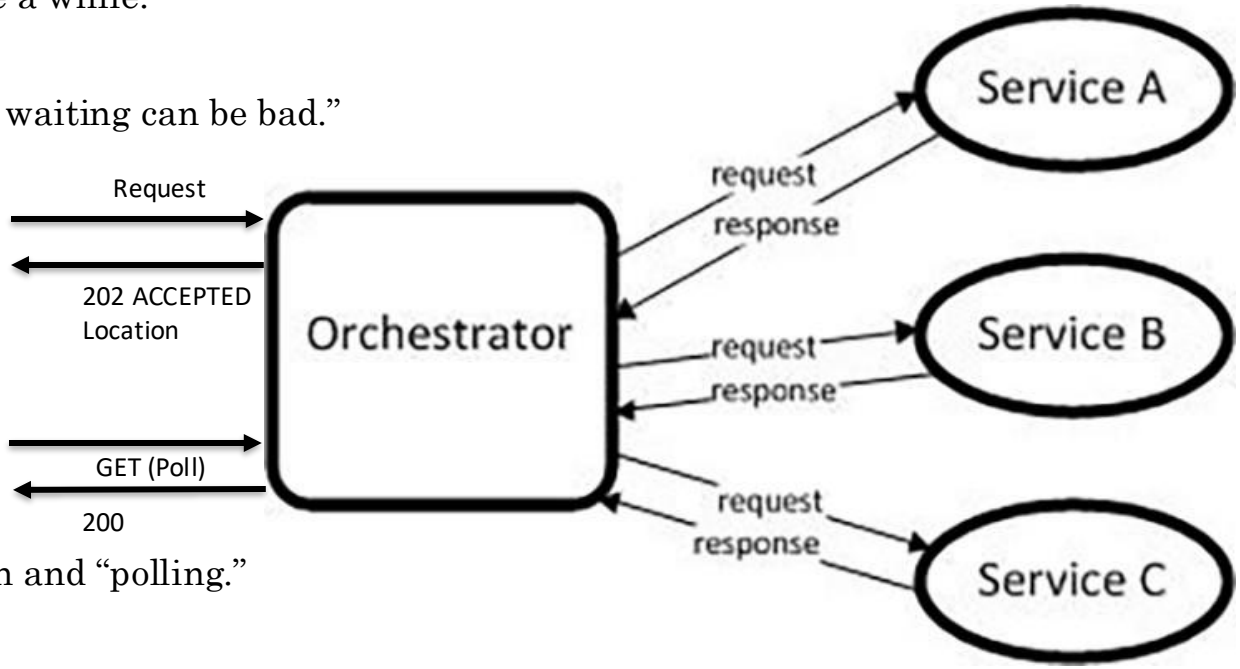


The choreography describes the interactions between multiple services, where as orchestration represents control from one party's perspective. This means that a **choreography** *differs* from an **orchestration** with respect to where the logic that controls the interactions between the services involved should reside.

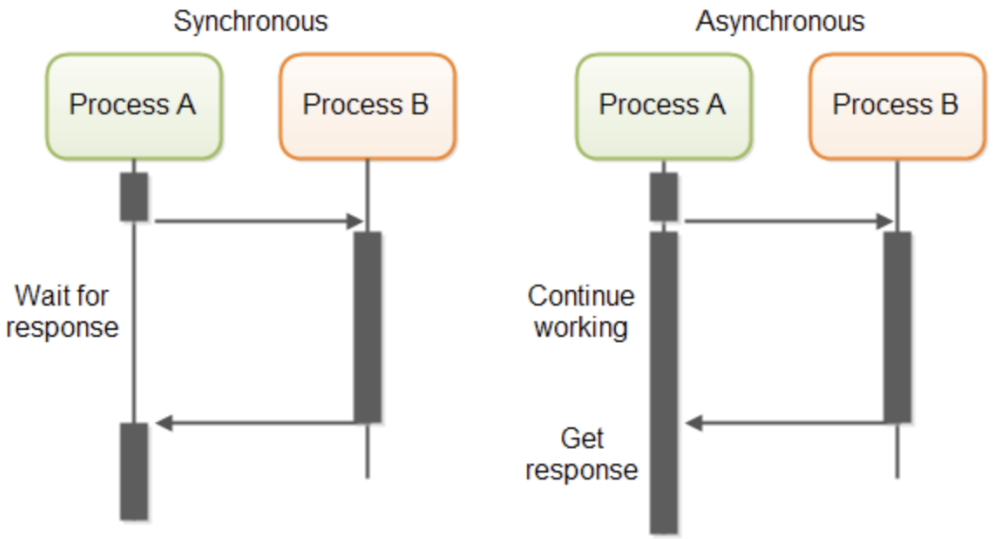
These are not formally, rigorously defined terms.

In Code: Composition and Asynchronous Method Invocation

- Composite operations can “take a while.”
- “Holding open connections and waiting can be bad.”
  - Consumes resources.
  - Connections can break.
  - … …
- There are two options:
  - Polling
  - Callbacks
- Our next pattern is composition and “polling.”
  Create a new user requires:
  - Updating the user database.
  - Verifying a submitted address.
  - Creating an account in the catalog service.



In Code: Service Orchestration/Composition

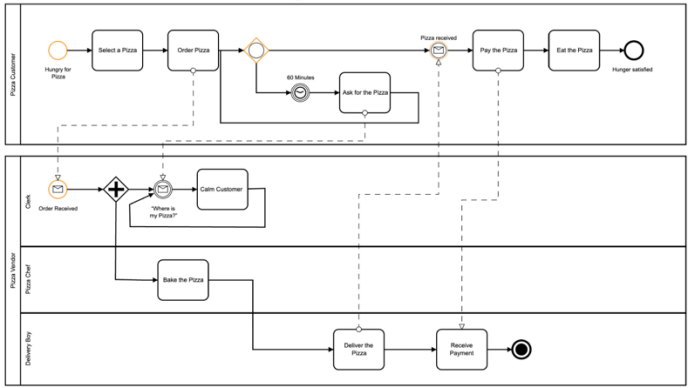


RESTFul HTTP calls can be implemented in both a synchronous and asynchronous fashion at an IO level.

- One call to a composite service
  - May drive calls to multiple other services.
  - Which in turn, may drive multiple calls to other services.
- Synchronous (Wait) and calling one at a time, in order is
  - Inefficient
  - Fragile
  - … …
- Asynchronous has benefits but can result in complex code.

Service Orchestration

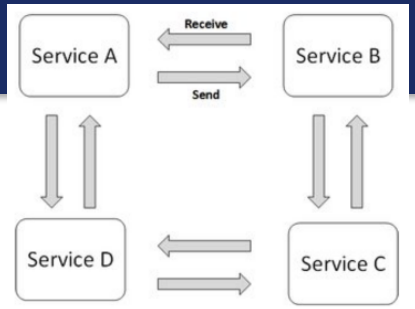
- Implementing a complex orchestration in code can become complex.
- High layer abstractions and tools have emerged, e.g. BPMN.
  - "Business Process Model and Notation (BPMN) is a graphical representation for specifying business processes in a business process model." expand in new window

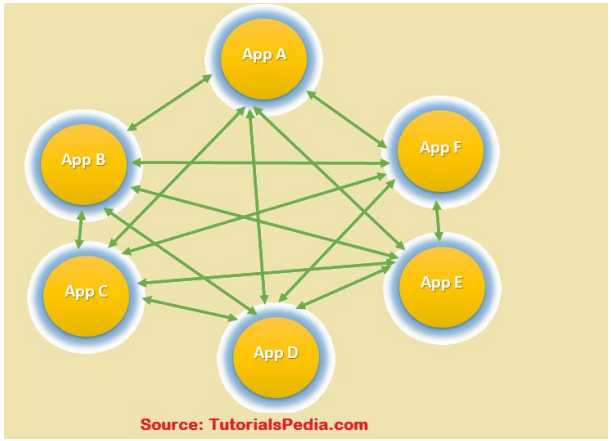


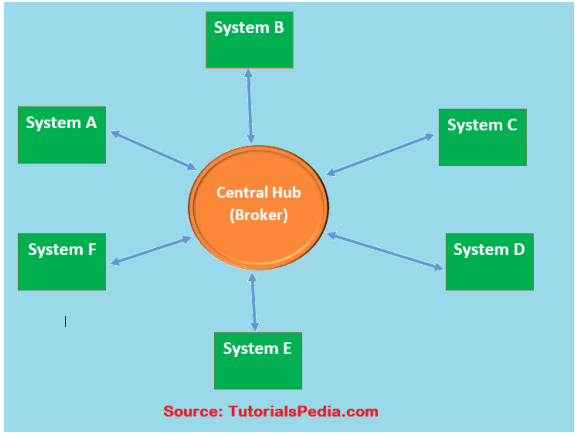
  - There are products for defining BPMN processes, generating code, executing, … …, e.g. Camunda: https://camunda.com/bpmn/
- There are other languages, tools, execution engines, … …

Choreography

- We saw the basic diagram, but …
  this is an anti-pattern and does not scale.







- But, how do you write the event driven microservices?
  - Well, you can just write code … …
  - Or use a state machines abstraction.
  - … …

Next Steps

- We will start implementing a composition microservice for your project.
- The initial implementation will use *orchestration.*
  - Synchronously calling the orchestrated microservices using “standard code.”
  - Asynchronously calling the orchestrated code using threads, promises, …
- We will then look at other models, e.g. EDA, MDP, State Machines, … …

```python
import asyncio

async def say_hello_async():
    await asyncio.sleep(2)
    print("Hello, Async World!")

asyncio.run(say_hello_async())
```

With `asyncio`, while we wait, the event loop can do other tasks, like checking

```python
import asyncio

async def say_hello_async():
    await asyncio.sleep(2)  # Simulates waiting for 2 seconds
    print("Hello, Async World!")

async def do_something_else():
    print("Starting another task...")
    await asyncio.sleep(1)  # Simulates doing something else for 1 second
    print("Finished another task!")

async def main():
    # Schedule both tasks to run concurrently
    await asyncio.gather(
        say_hello_async(),
        do_something_else(),
    )

asyncio.run(main())
```

https://medium.com/@moraneus/mastering-pythons-asyncio-a-practical-guide-0a673265cf04

Next Steps

- See for more complete example: https://github.com/donald-f-ferguson/W4153-Cloud-Computing-Base/blob/3d94111a1018b8b5aba9069a333f1e030d75d503/simple_examples/sync_async
- https://app.swaggerhub.com/search?owner=Columbia-Classes&visibility=PUBLIC

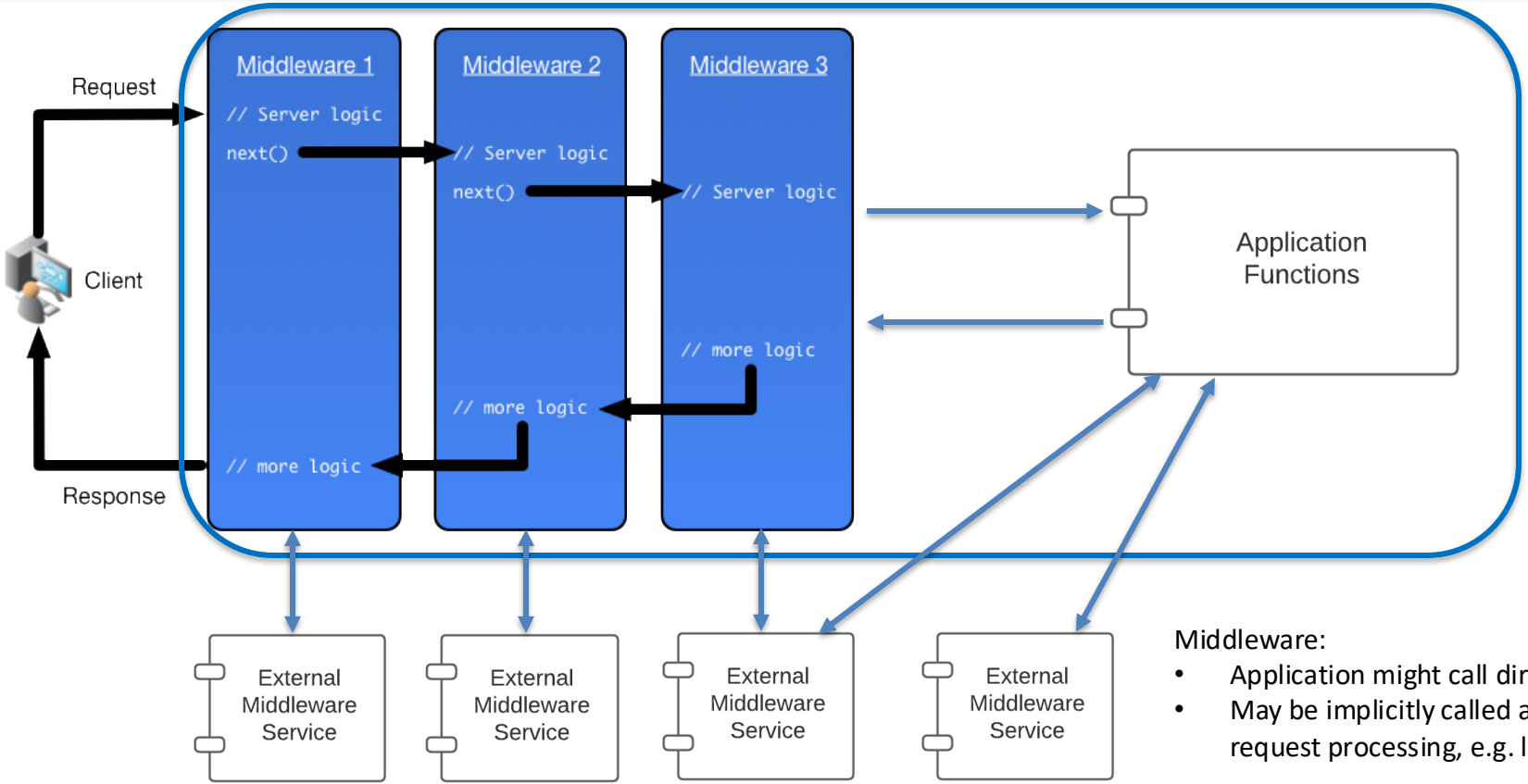
Middleware

What is middleware?

Middleware is software that lies between an operating system and the applications running on it. Essentially functioning as hidden translation layer, middleware enables communication and data management for distributed applications. It's sometimes called plumbing, as it connects two applications together so data and databases can be easily passed between the "pipe." Using middleware allows users to perform such requests as submitting forms on a web browser, or allowing the web server to return dynamic web pages based on a user's profile.

Common middleware examples include database middleware, application server middleware, message-oriented middleware, web middleware, and transaction-processing monitors. Each program typically provides messaging services so that different applications can communicate using messaging frameworks like simple object access protocol (SOAP), web services, representational state transfer (REST), and JavaScript object notation (JSON). While all middleware performs communication functions, the type a company chooses to use will depend on what service is being used and what type of information needs to be communicated. This can include security authentication, transaction management, message queues, applications servers, web servers, and directories. Middleware can also be used for distributed processing with actions occurring in real time rather than sending data back and forth.

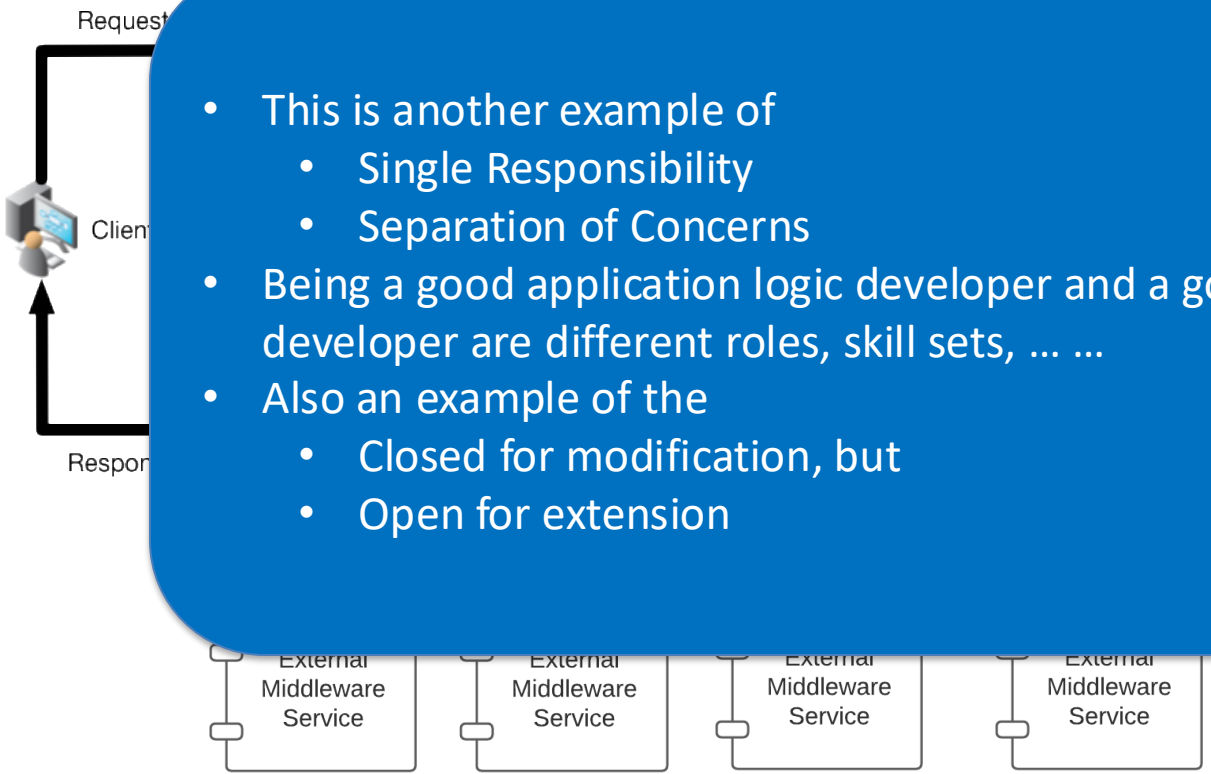
Middleware – Conceptual Model



Middleware:

- Application might call directly, e.g. DB
- May be implicitly called as part of request processing, e.g. logging.

Middleware – Conceptual Model



- This is another example of
  - Single Responsibility
  - Separation of Concerns
- Being a good application logic developer and a good middleware developer are different roles, skill sets, … …
- Also an example of the
  - Closed for modification, but
  - Open for extension

- Application might call directly, e.g. DB
- May be implicitly called as part of request processing, e.g. logging.

FastAPI Middleware (https://medium.com/@saverio3107/mastering-middleware-in-fastapi-from-basic-implementation-to-route-based-

What Does Middleware Do?

1. **Incoming Request:** When a request comes in, the middleware can examine and even modify parts of the request like cookies, headers, query parameters, etc., before it reaches the actual API service method. For example, it can validate the request, authenticate the user, or log the request details.
2. **Outgoing Response**: After the API service method has processed the request and generated a response, the middleware gets a chance to inspect and modify the response before it goes back to the client. This can include things like changing the response headers, transforming the content, and so on.

FastAPI Middleware

Middleware

https://fastapi.tiangolo.com/tutorial/middleware/
https://fastapi.tiangolo.com/advanced/middleware/

You can add middleware to **FastAPI** applications.

A "middleware" is a function that works with every **request** before it is processed by any specific *path operation*. And also with every **response** before returning it.

- It takes each **request** that comes to your application.
- It can then do something to that **request** or run any needed code.
- Then it passes the **request** to be processed by the rest of the application (by some *path operation*).
- It then takes the **response** generated by the application (by some *path operation*).
- It can do something to that **response** or run any needed code.
- Then it returns the **response**.

Example

- Walk through Medium Article

- https://medium.com/@saverio3107/mastering-middleware-in-fastapi-from-basic-implementation-to-route-based-strategies-d62eff6b5463

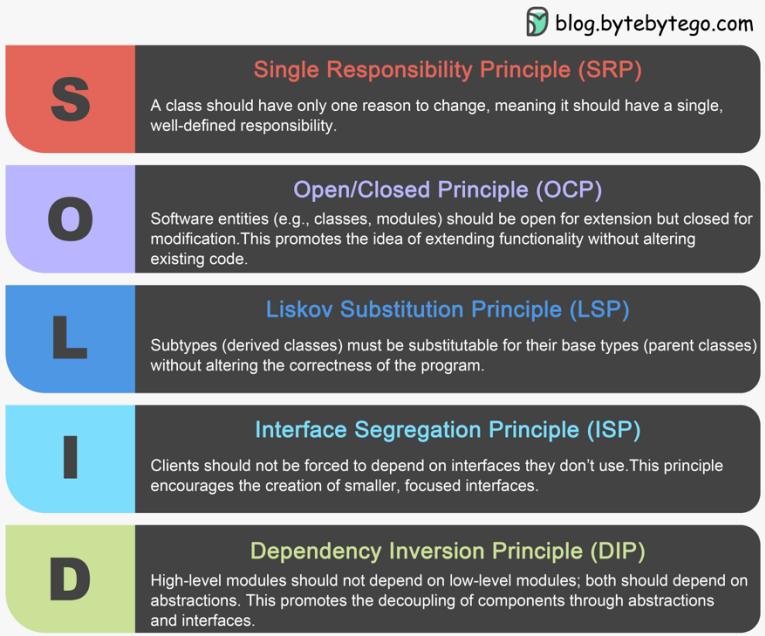
- This is the very simple version.
  There is a more sophisticated approach to middleware in FastAPI.
  https://fastapi.tiangolo.com/advanced/middleware/

12 Factor Apps
SOLID
Design Patterns

12 Factor Applications

https://dzone.com/articles/12-factor-app-principles-and-cloud-native-microser

SOLID Principles



Microservices Guidelines	SOLID Principles
Loosely coupled	Interface Segregation + Dependency Inversion
Testable	Single Responsibility + Dependency Inversion.
Composable	Single Responsibility + Open/close
Highly maintainable	Single Responsibility + Liskov Substitution + Interface Segregation
Independently deployable	Single Responsibility + Interface Segregation + Dependency Inversion
Capable of being developed by a small team	Single Responsibility + Interface Segregation

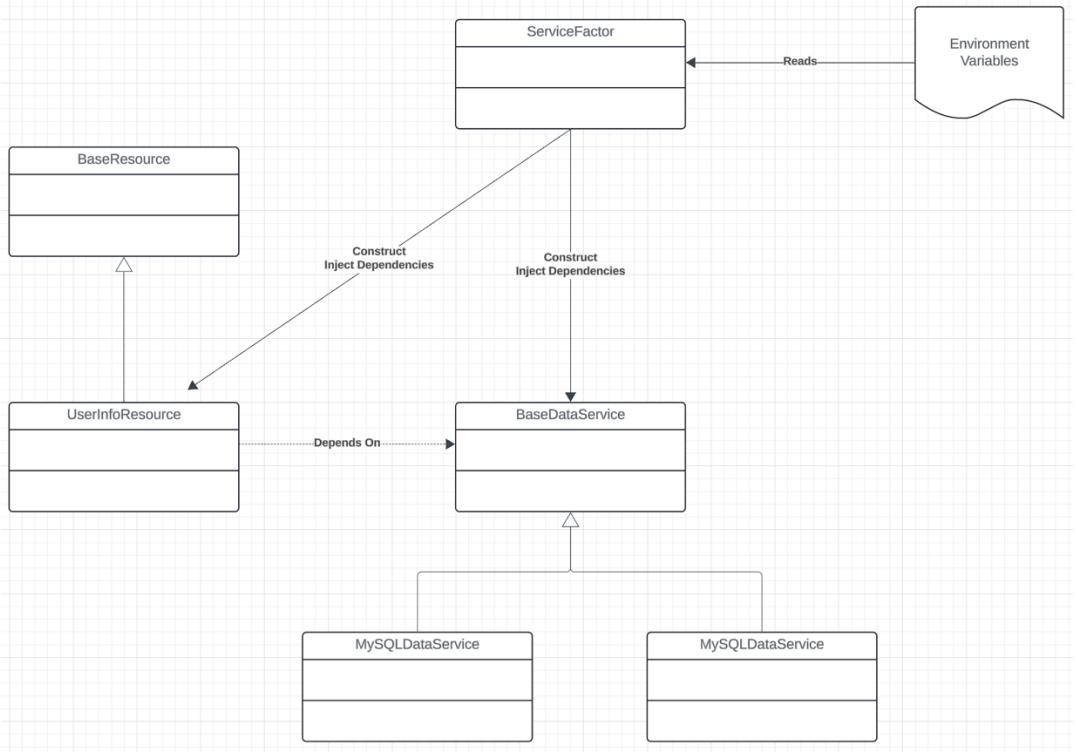
https://medium.com/@saurabh.engg.it/microservices-designing-effective-microservices-by-following-solid-design-principles-a995c3a033a0

This is a good overview article, and we will see patterns in later lectures.

- This slide and the preceding slides could have been a 60 slide presentation.
- If you put 3 SW architects into a room, you will get 15 design principles and patterns.
- Use common sense.

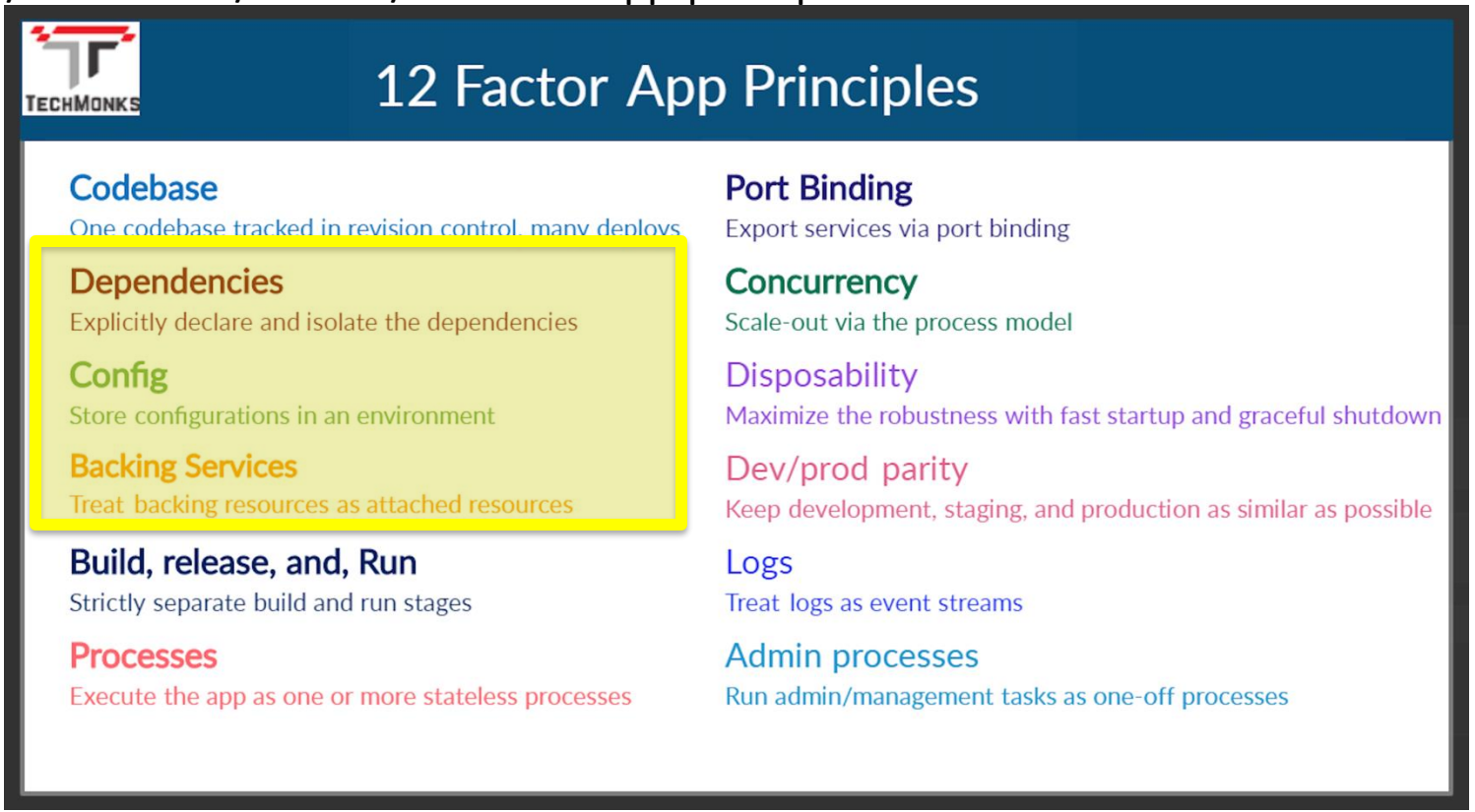
SW Architecture

- Use interfaces, or the language equivalent/approximation.
- A constructor receive a “config” object that is a dictionary of
  - Interface name.
  - Reference to implementation.
- Parameters and configuration properties are also in the config, e.g.
  - URLs
  - “Passwords”
  - Database and table names
  - … …
- A “service factory” creates the specific instances and configs,
  and wires it all together.



12 Factor Applications

https://dzone.com/articles/12-factor-app-principles-and-cloud-native-microser



Dependencies

II. Dependencies

Explicitly declare and isolate dependencies

Most programming languages offer a packaging system for distributing support libraries, such as CPAN for Perl or Rubygems for Ruby. Libraries installed through a packaging system can be installed system-wide (known as "site packages") or scoped into the directory containing the app (known as "vendoring" or "bundling").

A twelve-factor app never relies on implicit existence of system-wide packages. It declares all dependencies, completely and exactly, via a *dependency declaration* manifest. Furthermore, it uses a *dependency isolation* tool during execution to ensure that no implicit dependencies "leak in" from the surrounding system. The full and explicit dependency specification is applied uniformly to both production and development.

For example, Bundler for Ruby offers the `Gemfile` manifest format for dependency declaration and `bundle exec` for dependency isolation. In Python there are two separate tools for these steps – Pip is used for declaration and Virtualenv for isolation. Even C has Autoconf for dependency declaration, and static linking can provide dependency isolation. No matter what the toolchain, dependency declaration and isolation must always be used together – only one or the other is not sufficient to satisfy twelve-factor.

One benefit of explicit dependency declaration is that it simplifies setup for developers new to the app. The new developer can check out the app's codebase onto their development machine, requiring only the language runtime and dependency manager installed as prerequisites. They will be able to set up everything needed to run the app's code with a deterministic *build command*. For example, the build command for Ruby/Bundler is `bundle install`, while for Clojure/Leiningen it is `lein deps`.

Twelve-factor apps also do not rely on the implicit existence of any system tools. Examples include shelling out to ImageMagick or `curl`. While these tools may exist on many or even most systems, there is no guarantee that they will exist on all systems where the app may run in the future, or whether the version found on a future system will be compatible with the app. If the app needs to shell out to a system tool, that tool should be vendored into the app.

- Most application frameworks have a dependency declaration concept.
- I have been showing examples with:
  - Python import
  - requirements.txt
  - etc.
- My examples also show encapsulating libraries and APIs with Python classes.
- II. Dependencies handles "the code," but we still need to handle the instance. There is a difference between, e.g.
  - pymysql
  - A specific connection.

III. Configurations

III. Config

Store config in the environment

An app's *config* is everything that is likely to vary between deploys (staging, production, developer environments, etc). This includes:

- Resource handles to the database, Memcached, and other backing services
- Credentials to external services such as Amazon S3 or Twitter
- Per-deploy values such as the canonical hostname for the deploy

Apps sometimes store config as constants in the code. This is a violation of twelve-factor, which requires **strict separation of config from code**. Config varies substantially across deploys, code does not.

A litmus test for whether an app has all config correctly factored out of the code is whether the codebase could be made open source at any moment, without compromising any credentials.

Note that this definition of "config" does **not** include internal application config, such as `config/routes.rb` in Rails, or how code modules are connected in Spring. This type of config does not vary between deploys, and so is best done in the code.

Another approach to config is the use of config files which are not checked into revision control, such as `config/database.yml` in Rails. This is a huge improvement over using constants which are checked into the code repo, but still has weaknesses: it's easy to mistakenly check in a config file to the repo; there is a tendency for config files to be scattered about in different places and different formats, making it hard to see and manage all the config in one place. Further, these formats tend to be language- or framework-specific.

The twelve-factor app stores config in *environment variables* (often shortened to *env vars* or *envs*). Env vars are easy to change between deploys without changing any code; unlike config files, there is little chance of them being checked into the code repo accidentally; and unlike custom config files, or other config mechanisms such as Java System Properties, they are a language- and OS-agnostic standard.

Another aspect of config management is grouping. Sometimes apps batch config into named groups (often called "environments") named after specific deploys, such as the `development`, `test`, and `production` environments in Rails. This method does not scale cleanly: as more deploys of the app are created, new environment names are necessary, such as `staging` or `qa`. As the project grows further, developers may add their own special environments like `joes-staging`, resulting in a combinatorial explosion of config which makes managing deploys of the app very brittle.

In a twelve-factor app, env vars are granular controls, each fully orthogonal to other env vars. They are never grouped together as "environments", but instead are independently managed for each deploy. This is a model that scales up smoothly as the app naturally expands into more deploys over its lifetime.

- Each of the deployment environments we have seen has a method for setting environment variables.
  - https://docs.aws.amazon.com/cloud9/latest/user-guide/env-vars.html
  - https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/environments-cfg-softwaresettings.html
  - https://cloud.google.com/functions/docs/configuring/env-var
- CI/CD, GitHub actions, etc. can integrate environment variables with the development process.
- Vaults and secrets managers are a better approach for passwords.

IV. Backing Services

IV. Backing services

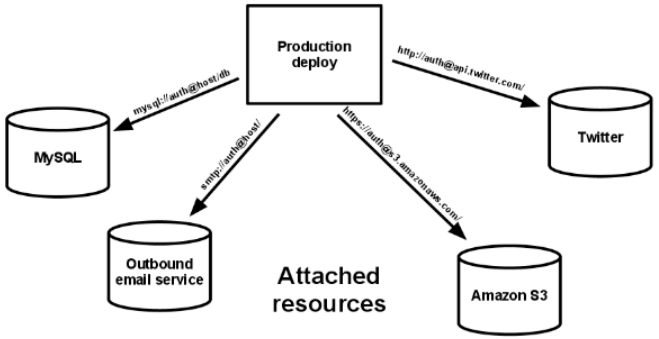
Treat backing services as attached resources

A *backing service* is any service the app consumes over the network as part of its normal operation. Examples include datastores (such as MySQL or CouchDB), messaging/queueing systems (such as RabbitMQ or Beanstalkd), SMTP services for outbound email (such as Postfix), and caching systems (such as Memcached).

Backing services like the database are traditionally managed by the same systems administrators who deploy the app's runtime. In addition to these locally-managed services, the app may also have services provided and managed by third parties. Examples include SMTP services (such as Postmark), metrics-gathering services (such as New Relic or Loggly), binary asset services (such as Amazon S3), and even API-accessible consumer services (such as Twitter, Google Maps, or Last.fm).

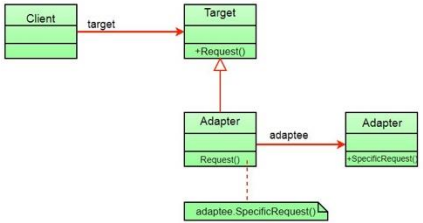
The code for a twelve-factor app makes no distinction between local and third party services. To the app, both are attached resources, accessed via a URL or other locator/credentials stored in the config. A deploy of the twelve-factor app should be able to swap out a local MySQL database with one managed by a third party (such as Amazon RDS) without any changes to the app's code. Likewise, a local SMTP server could be swapped with a third-party SMTP service (such as Postmark) without code changes. In both cases, only the resource handle in the config needs to change.

Each distinct backing service is a *resource*. For example, a MySQL database is a resource; two MySQL databases (used for sharding at the application layer) qualify as two distinct resources. The twelve-factor app treats these databases as *attached resources*, which indicates their loose coupling to the deploy they are attached to.



Resources can be attached to and detached from deploys at will. For example, if the app's database is misbehaving due to a hardware issue, the app's administrator might spin up a new database server restored from a recent backup. The current production database could be detached, and the new database attached – all without any code changes.

- My code has shown encapsulating backing services with a shallow adaptor.
- This is an example of the *adaptor pattern.*



- You seen examples of things like AbstractBaseDataService, … …
- The approach is also an application of several aspects of SOLID. (https://en.wikipedia.org/wiki/SOLID)

Structural Composition

- We can use four patterns/principles for service composition ”in code.”
  - Dependency Injection
  - Service Factory
  - Service Locator
  - Metadata in the environment for configuration.

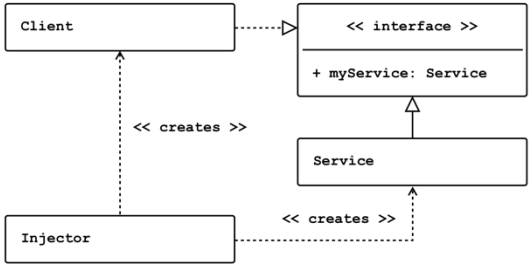
- I did some simple examples in https://github.com/donald-f-ferguson/E6156-Public-Examples/tree/master/old_examples/simple_pattern_examples.

- Or actually, I asked ChatGPT to produce simple examples.

- And, these are conceptual, and I am not super happy with them.

Dependency Injection

- Concepts (https://en.wikipedia.org/wiki/Dependency_injection):
  - In software engineering, dependency injection is a technique in which an object receives other objects that it depends on, called dependencies.
  - The intent behind dependency injection is to achieve separation of concerns of construction and use of objects. This can increase readability and code reuse.



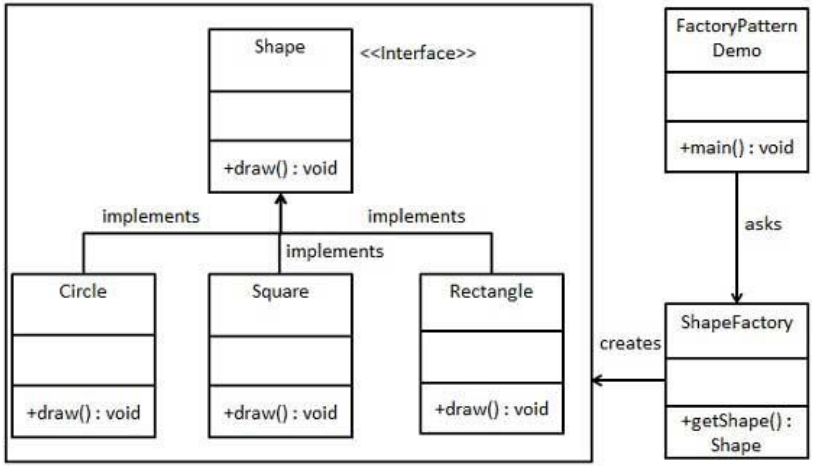

```
class UserResource(BaseApplicationResource):

    def __init__(self, config_info):
        super().__init__(config_info)
```

- There is no single approach. There are several frameworks, tools, etc.
- Like many other things, it is easy to get carried away and become dogmatic.
- I follow a single approach:
  - A Context class converts environment and other configuration and provides to application.
  - The top-level application injects a config_info object into services.

Service Factory

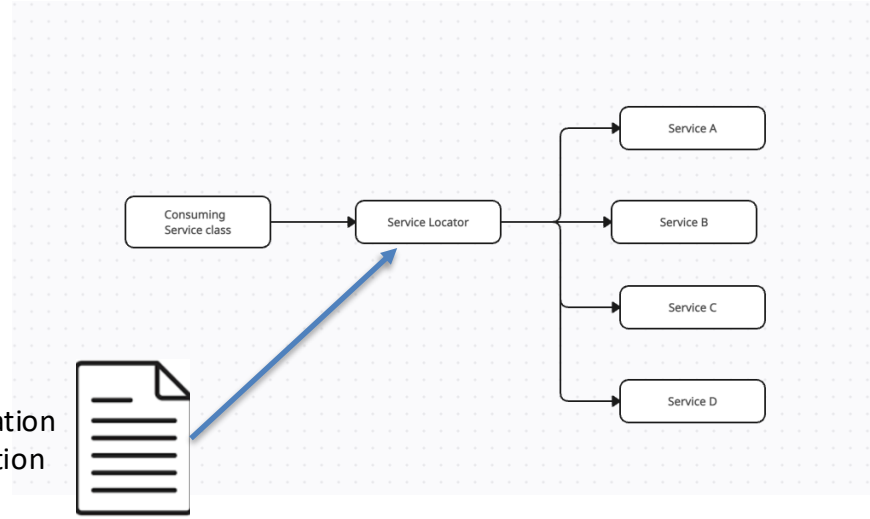
- "In class-based programming, the factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor." (https://en.wikipedia.org/wiki/Factory_method_pattern)
- You will sometimes see me use this for
  - Abstraction between a REST resource impl. and the data service.
  - Allows changing the database service, model, etc. without modifying the code.
  - Concrete implementation choices are configured via properties, metadata, …
- You can use in many situations.



https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

Service Locator

- “The service locator pattern is a design pattern used in software development to encapsulate the processes involved in obtaining a service with a strong abstraction layer. This pattern uses a central registry known as the "service locator", which on request returns the information necessary to perform a certain task. Proponents of the pattern say the approach simplifies component-based applications where all dependencies are cleanly listed at the beginning of the whole application design, …” (https://en.wikipedia.org/wiki/Service_locator_pattern)



Interesting References

Interesting Articles

- https://medium.com/@cihanelibol99/authorization-architecture-in-microservices-68085d31f1e1
- https://medium.com/@sylvain.tiset/top-10-microservices-design-patterns-you-should-know-1bac6a7d6218

Where Are We Going With All Of This?