Software Architecture 101

From Requirements to a Software Product

Matthias Linhuber

matthias@linhuber.org mtze.me github.com/Mtze

Who am I?

- Matthias Linhuber
- Doctoral Student at TUM
- Educator at heart
- Software and Infrastructure Architect
- Research Areas
  - Container-based Software Engineering
  - Infrastructure Orchestration
  - Scaling Education Technology

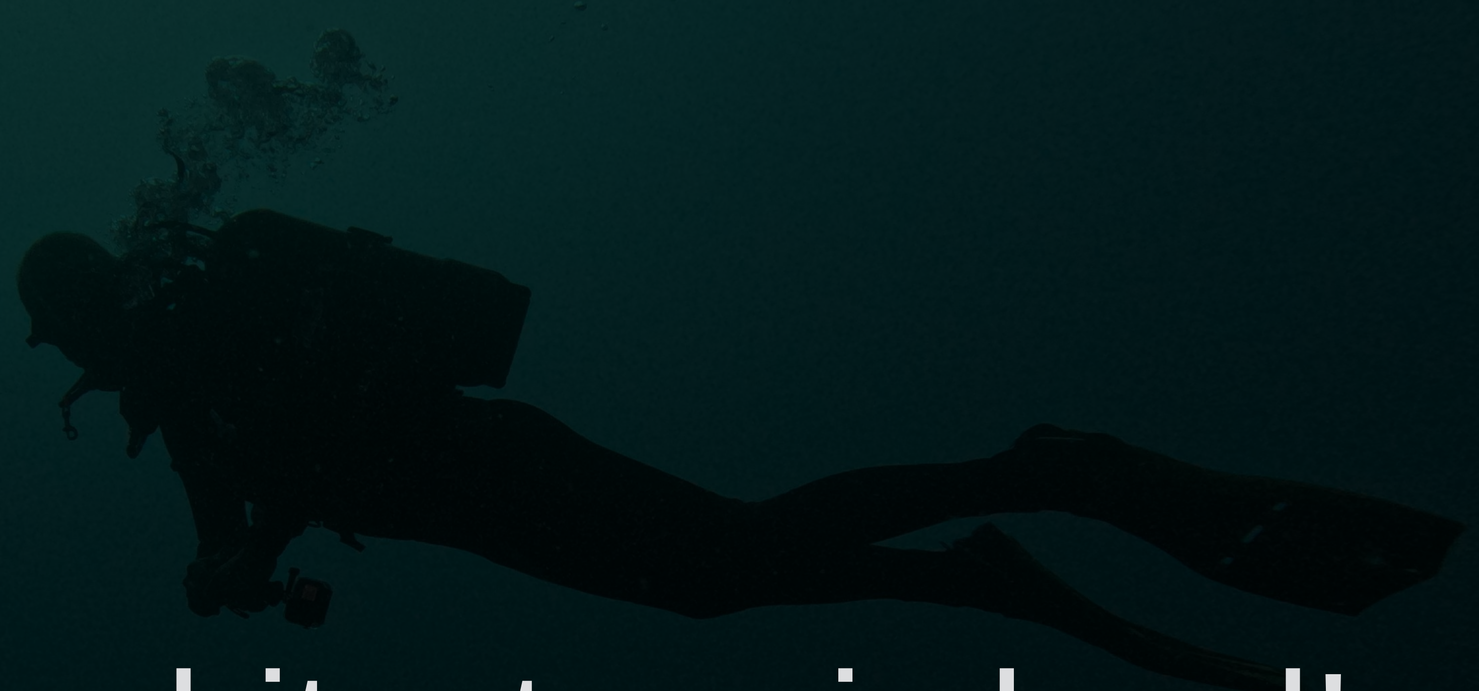
Context and Learning Goals

- You already know
  - Requirements Engineering
  - User Stories
  - Non-functional Requirements / Constraints
- After this workshop you
  - Understand the activities to build an Architecture
  - Understand the concepts of **coupling** and **cohesion**
  - Understand the importance of **Design Goals**
  - Create a **Top Level Architecture**
  - Create a **Subsystem Decomposition**
  - Create an **API Design in a Service Based Architecture**

Recap - Requirements Engineering

- Application Domain
  - Understand the **Problem** of the user
  - Understand involved **Objects**
  - Understand the **Environment Terminology**

- Solution Domain
  - Understand possible **Solutions** to the problem
  - Argue about **Technologies**, implementation **Styles** and **Techniques**
  - Decide what to do!



Software Architecture is hard!

Takes practice!

Software Architecture ~~is hard~~!

So let’s take the plunge…

Discussion

This is the sign for you to take action!

- What topics need to be discussed/defined in an Architecture?
- Why are they important?

Software Architecture - Definitions

Architecture is the fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution.

The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

Software architecture is a set of design decisions that are expensive to change.

Architecture Activities

- We bridge the gap between the **problem** and the **system**
- Resulting artifacts:
  - **Design Goals:** Describe the qualities of the system
  - **Subsystem Decomposition:** Defines subsystem, their service and relationships
  - **Hardware Software Mapping:** Define the deployment of Subsystems
  - **Data Management:** Defines which, where and how data should be persistent
  - **Access Control:** Defines who can access which data when
  - **Boundary Conditions:** Defines how to start, stop or recover the System

Design Goals

Design Goals

- Also called **Quality Attributes**
- Define how the system should **behave**, not what it should do
- Influence architecture more than functional requirements

Design goals are the key drivers of architectural decisions.

“You can implement the same features in a hundred ways. Design goals decide which one is right.”

Design Goals

- Starting Point: **All NFRs from the RE process are Design Goals**
- Additional Design Goals

Non-exhaustive!

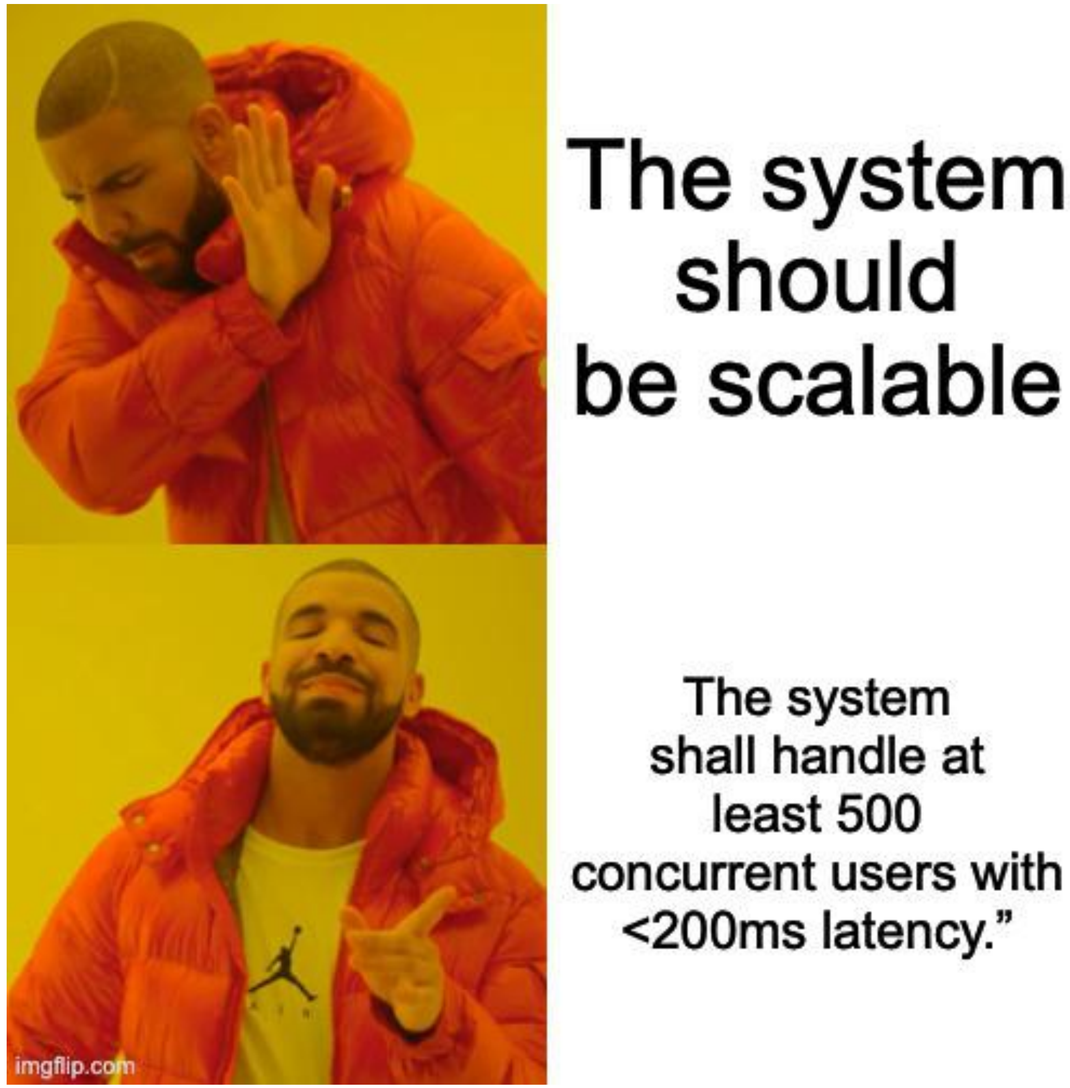
  - **Reusability:** Usable in related applications with minor modification?
  - **Scalability:** How many users should the system support?
  - **Elasticity:** Should the system scale dynamically?
  - **Robustness:** Mitigates errors and recovers from them?
  - **Security:** Do we have sensitive information that needs attention?
  - **Modifiability:** How easy can the system be adjusted?
  - **Availability:** Do we need to give uptime guarantees?

Design Goals

- **Design trade-offs** and **conflicting requirements**
  - **Functionality vs. Usability:** Rich features often make interfaces more complex or overwhelming to users.
  - **Performance vs. Modifiability:** Highly optimized code is often harder to understand or refactor.
  - **Flexibility vs. Complexity:** Making a system too configurable (via plugins, XML, DSLs) makes it hard to maintain or onboard.
- Design Goals have to be **prioritized**
  - MoSCoW Method (Must, Should, Could, Won't)

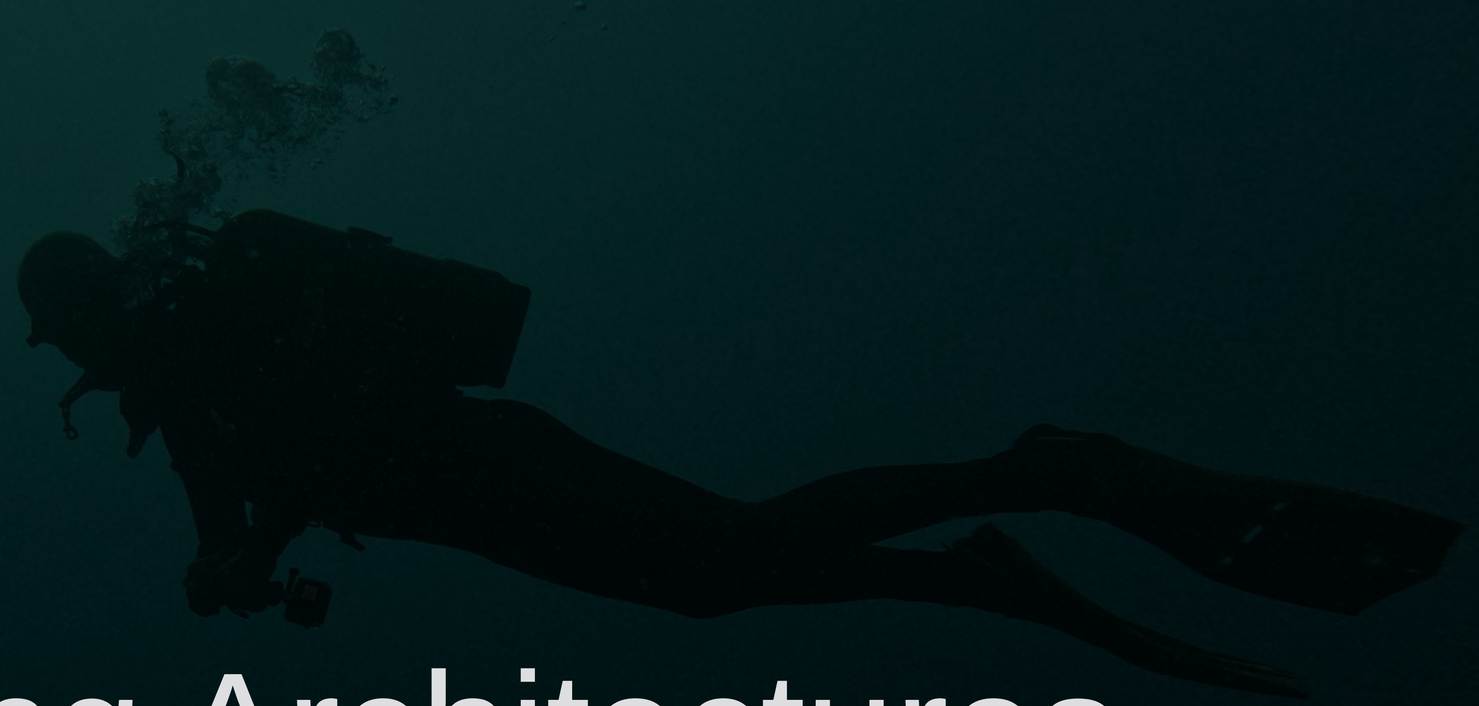
Discuss with your Product Owner / Steak holders

Design Goals have to be measurable



Example: Travel Booking System

- **Design Goal**
  - The system should be implemented as micro service architecture to allow independent deployability of software components
  - All system APIs have to be versioned



Modeling Architectures

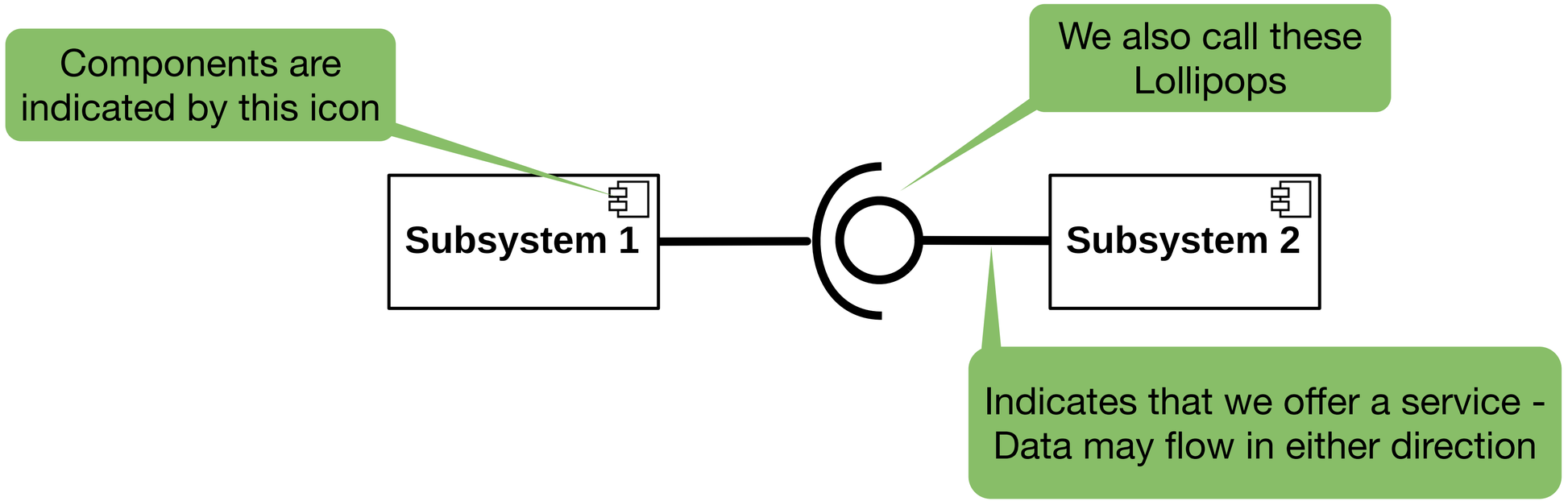
Modeling Architectures

All models are wrong, but some are useful!

- Why modeling?
  - **Communicate** design at the right level of abstraction
  - **Align team understanding** across stakeholders
  - Enable analysis: interfaces, dependencies, deployment
- Modeling Tools: UML, C4, …
  - We will focus on UML Component Diagrams

UML Component Diagram

- A **static view** of the system's component **structure**
- Focuses on **components**, provided/required **interfaces**, and **dependencies**



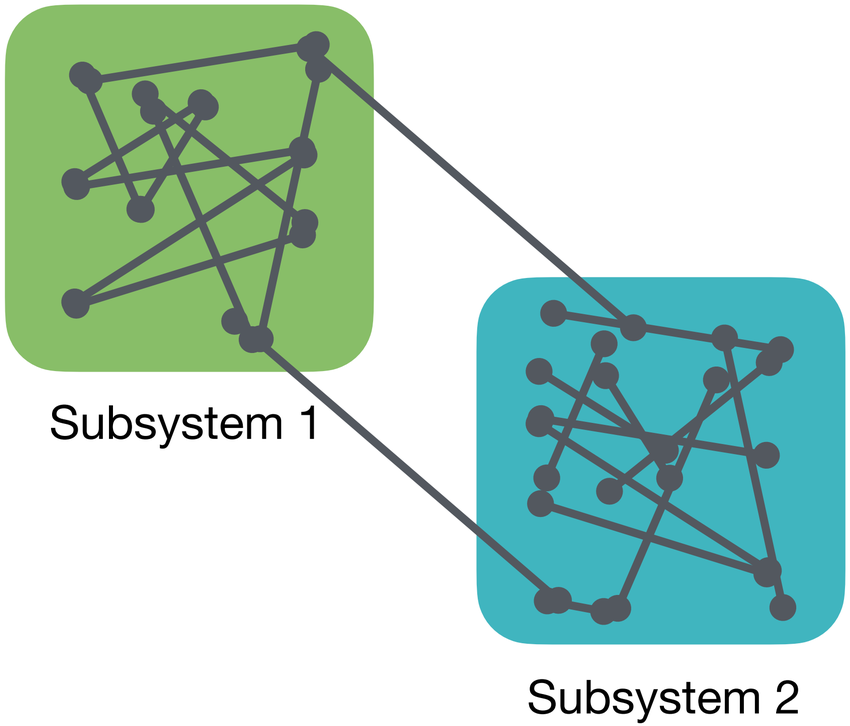
Subsystem Decomposition

- UML **Component Diagram**
- Place for **architectural decisions**
- Architectural Styles
  - Layered Architecture
  - Client Server Architecture
  - Peer to Peer Architecture
  - REST Architecture
  - Micro Service Architecture
  - ...

Coupling and Cohesion

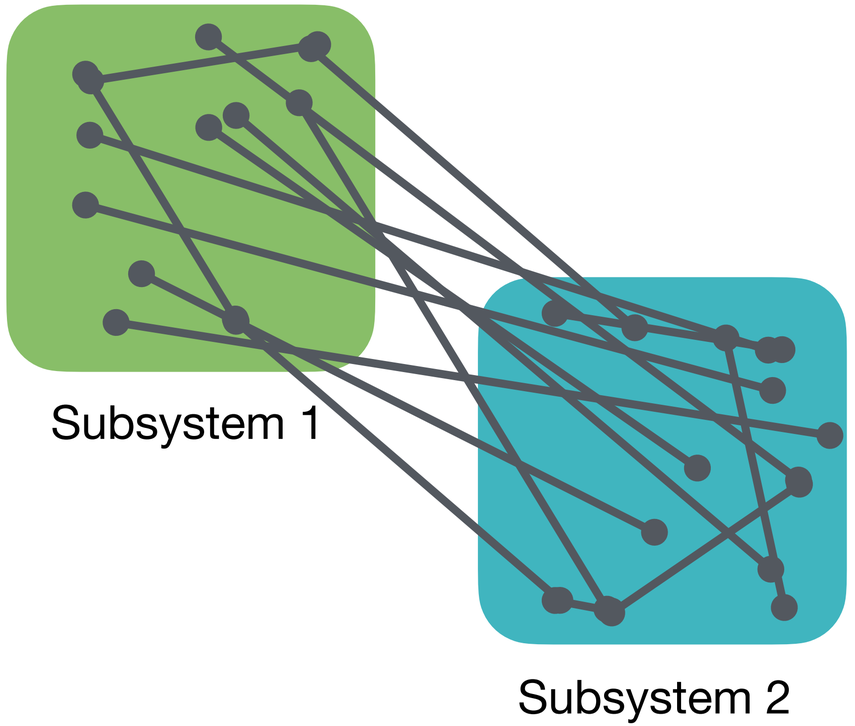
- We aim to have **low coupling and high cohesion**
- **Coupling** describes the number of dependencies **between** two subsystems
- **Cohesion** describes number of dependencies **within** a subsystem

Coupling and Cohesion



Low Coupling, High Cohesion

😁



High Coupling, Low Cohesion

🤬

From Requirements to Architecture

Top Level Architecture

- Objective:
  - **Mental model** of the solution domain
  - Explain the **overall architecture** of the system
  - Explain the **data flows** in your system
  - Intermediate step to the Subsystem Decomposition
  - “System Architecture for semi/non-technical persons”
- Informal Model
  - No strict rules
  - **Find “right” level of abstraction**

Designing a Top Level Architecture

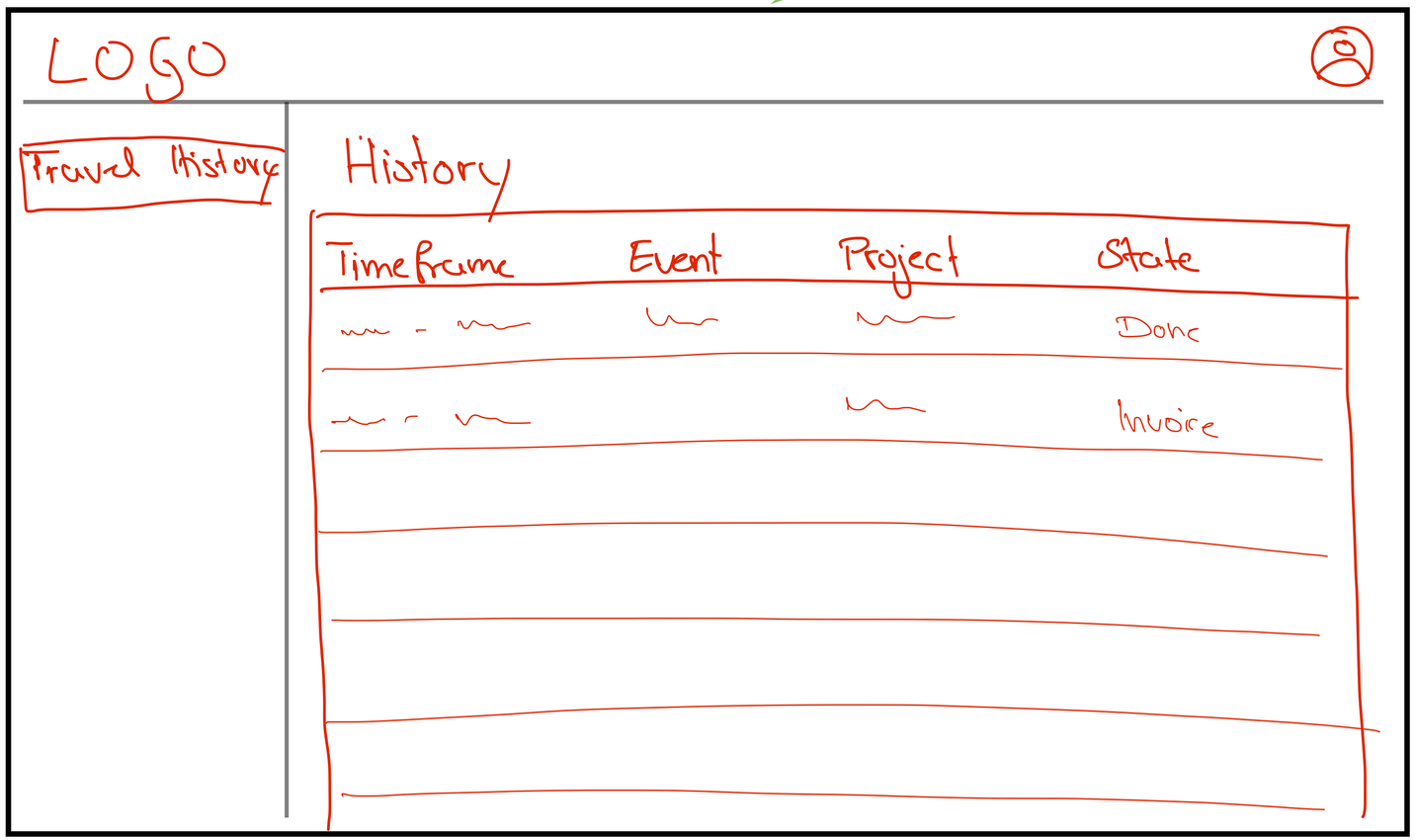
1. **Pick a User Story**
2. Look at participating objects in the AOM
3. Group related objects to one system
4. Model the dataflow
5. Specify the required Services
6. Adapt to comply with Design Goals

Example: Travel Booking System

- **User Story:**
  - As an Employee, I want to view my past trips and expenses so that I can reuse data for new travel requests.
  - Acceptance Criteria
    - Employee sees a list of past trips with expense totals.
    - User can duplicate previous trip details.

Example: Travel Booking System

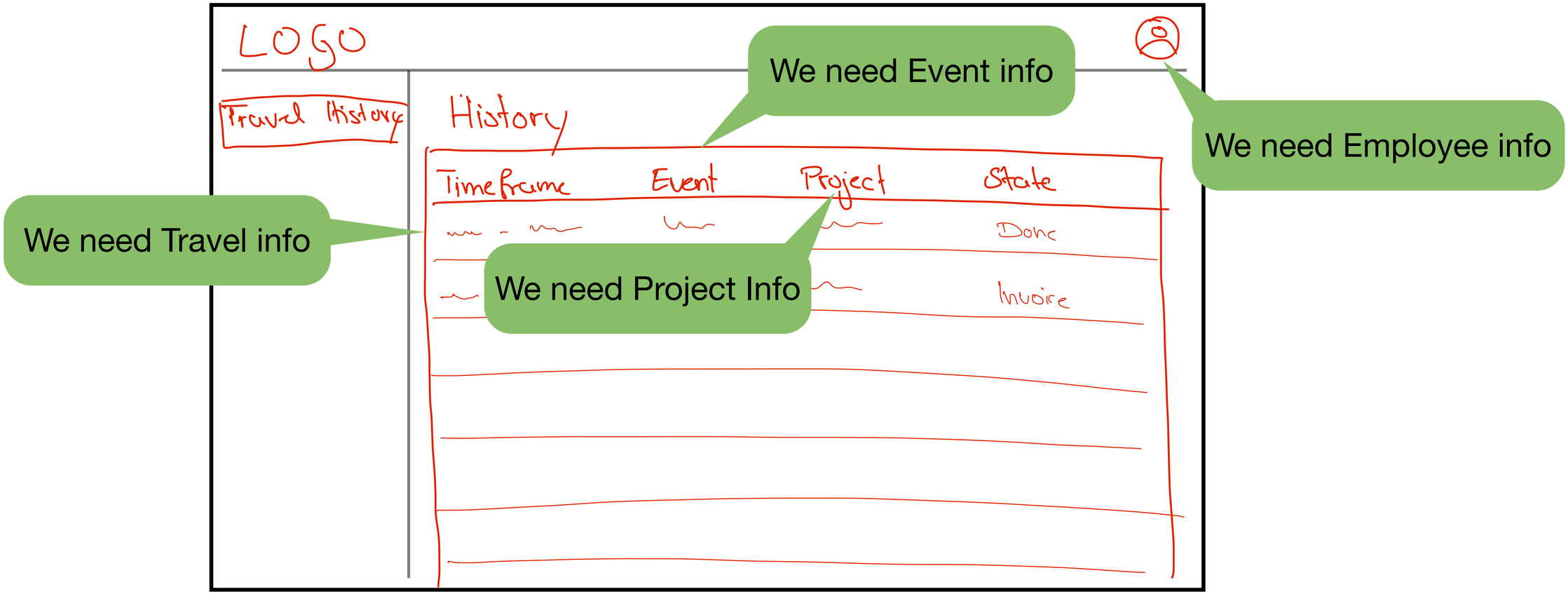
We should have a good idea how this feature should look like



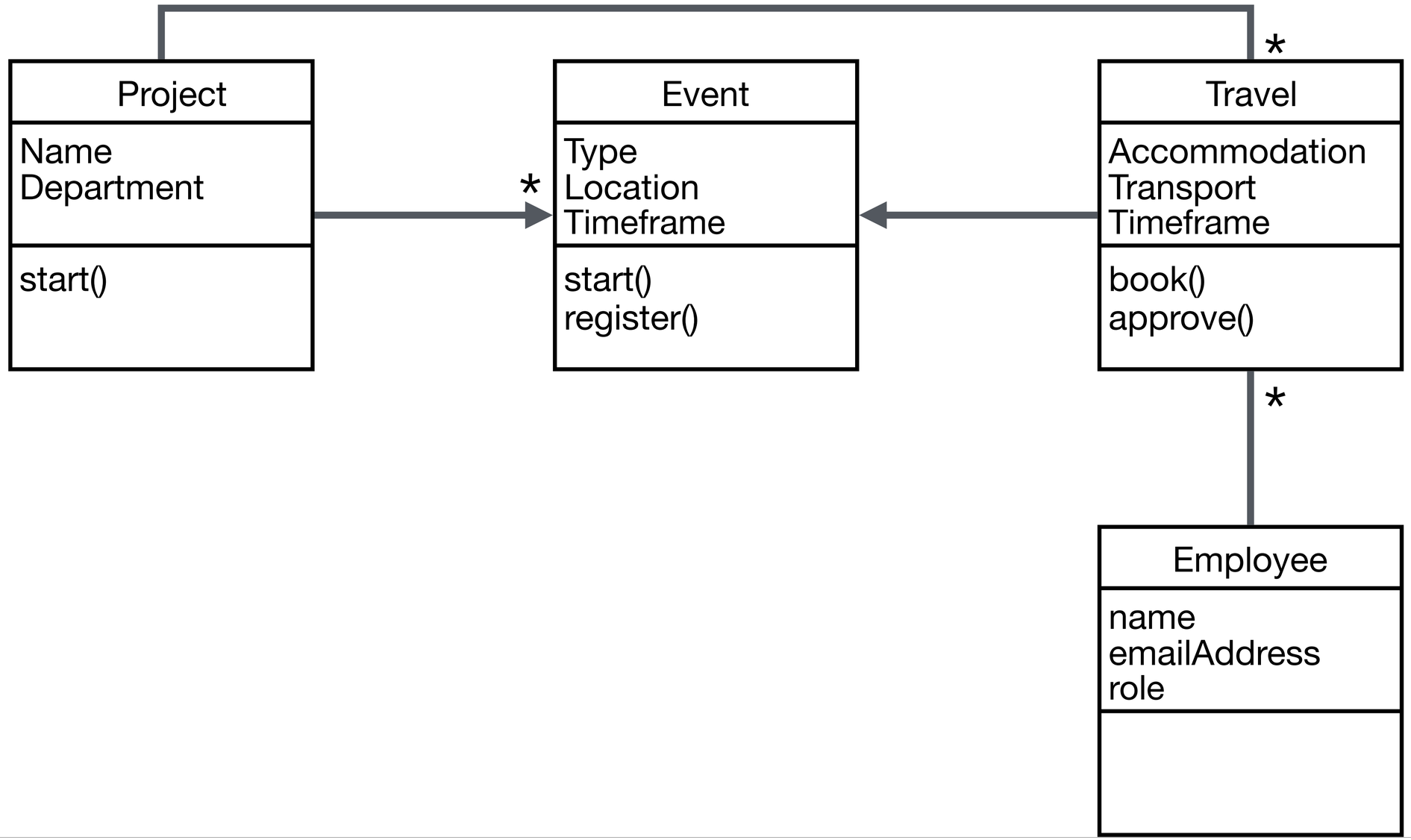
Designing a Top Level Architecture

1. Pick a User Story
2. **Look at participating objects in the AOM**
3. **Group related objects to one system**
4. Model the dataflow
5. Specify the required Services
6. Adapt to comply with Design Goals

Example: Travel Booking System



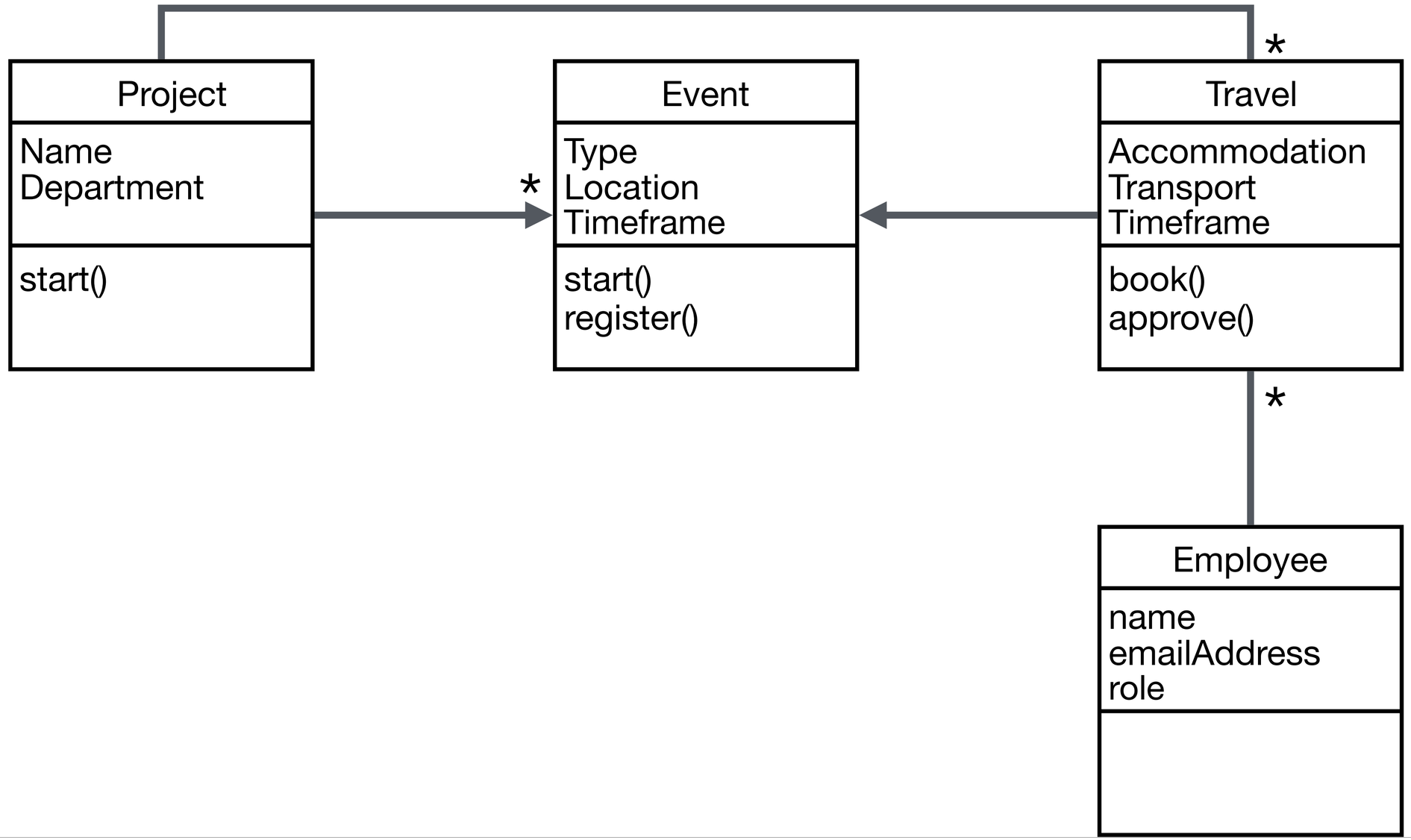
Example: Travel Booking System



Designing a Top Level Architecture

1. Pick a User Story
2. Look at participating objects in the AOM
3. Group related objects to one system
4. **Model the dataflow**
5. **Specify the required Services**
6. Adapt to comply with Design Goals

Example: Travel Booking System

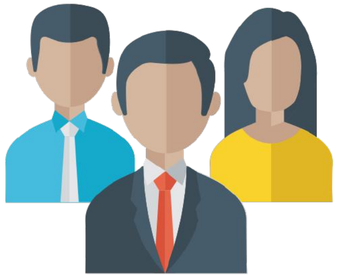


Example: Travel Booking System

Project

Event

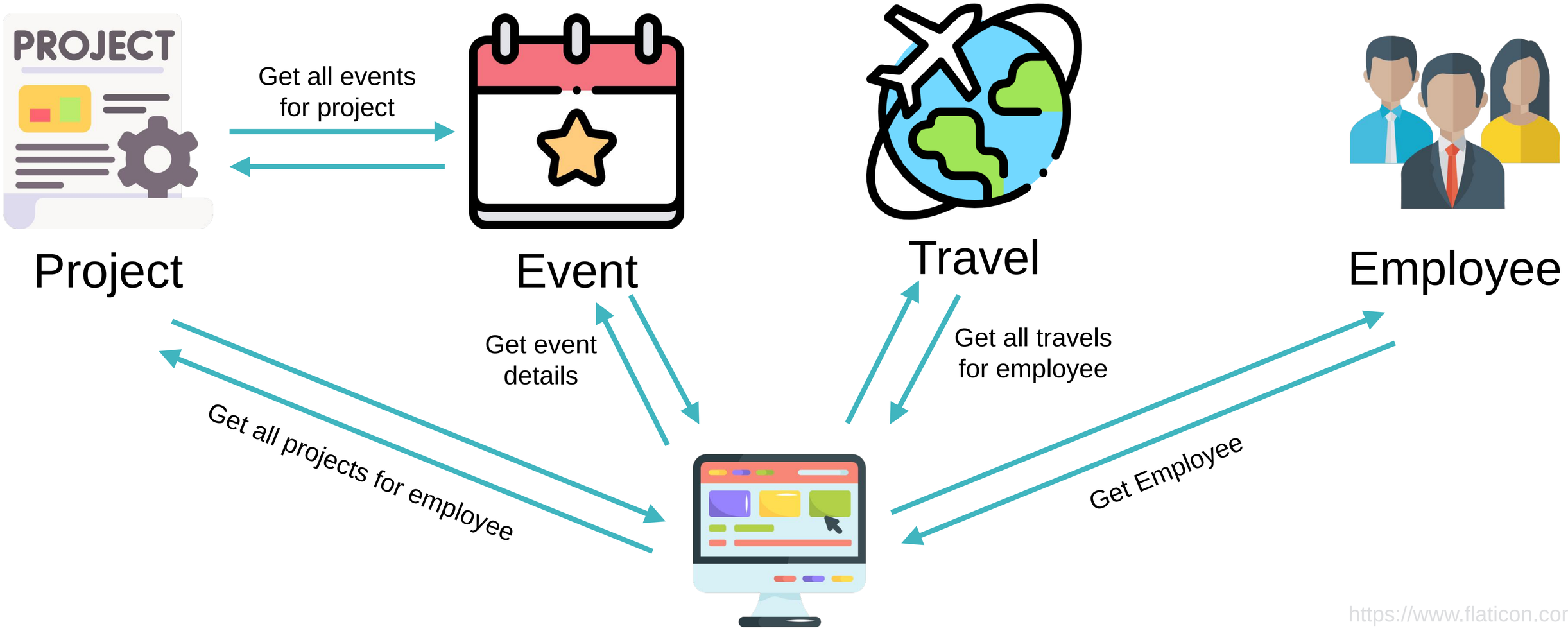
Travel



Employee

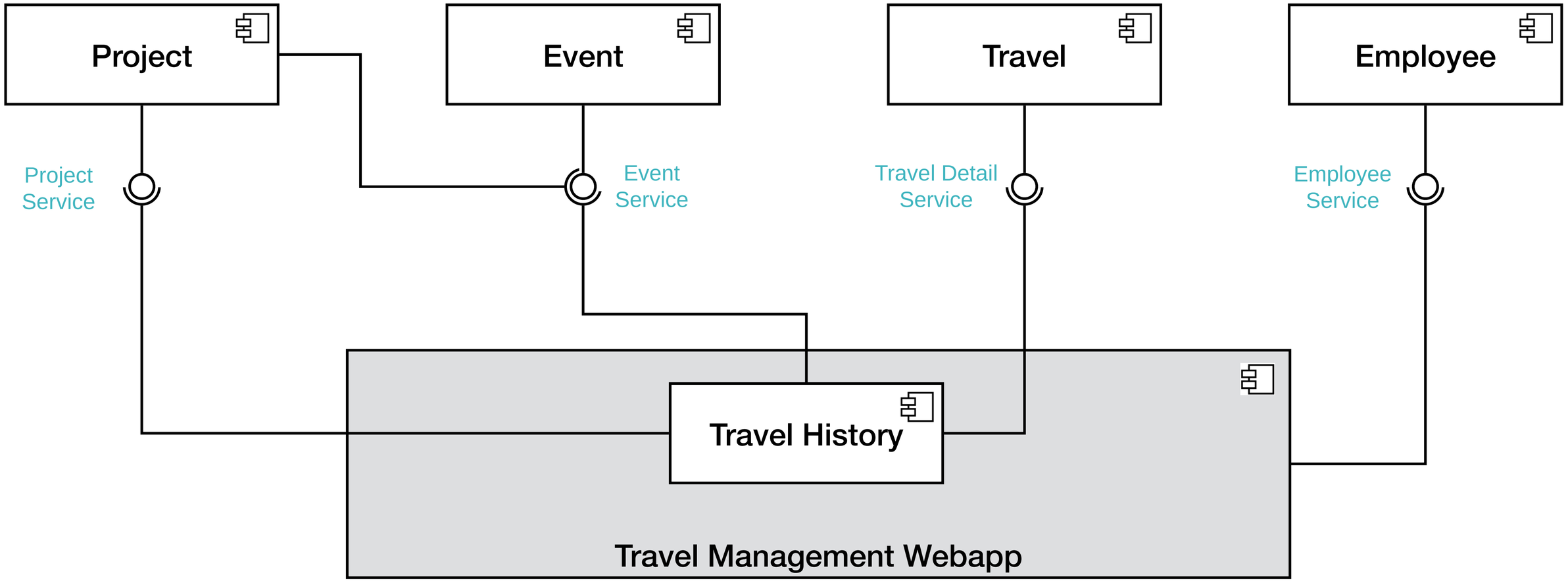
https://www.flaticon.com/

Example: Travel Booking System



https://www.flaticon.com/

Example: Travel Booking System



Designing a Top Level Architecture

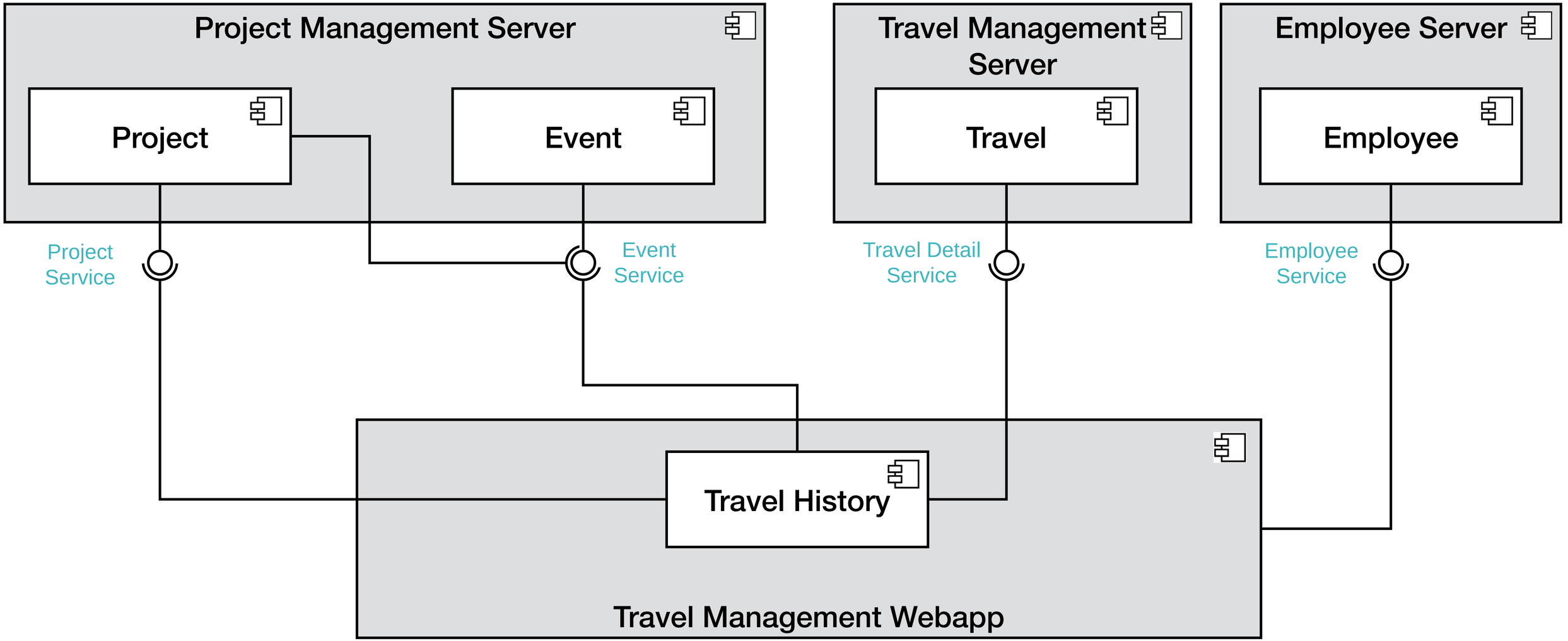
1. Pick a User Story
2. Look at participating objects in the AOM
3. Group related objects to one system
4. Model the dataflow
5. Specify the required Services
6. **Adapt to comply with Design Goals**

Example: Travel Booking System

- **Design Goal**
  - The system should be implemented as micro service architecture to allow independent deployability of software components
  - All system APIs have to be versioned

We decided to use a micro service architecture!

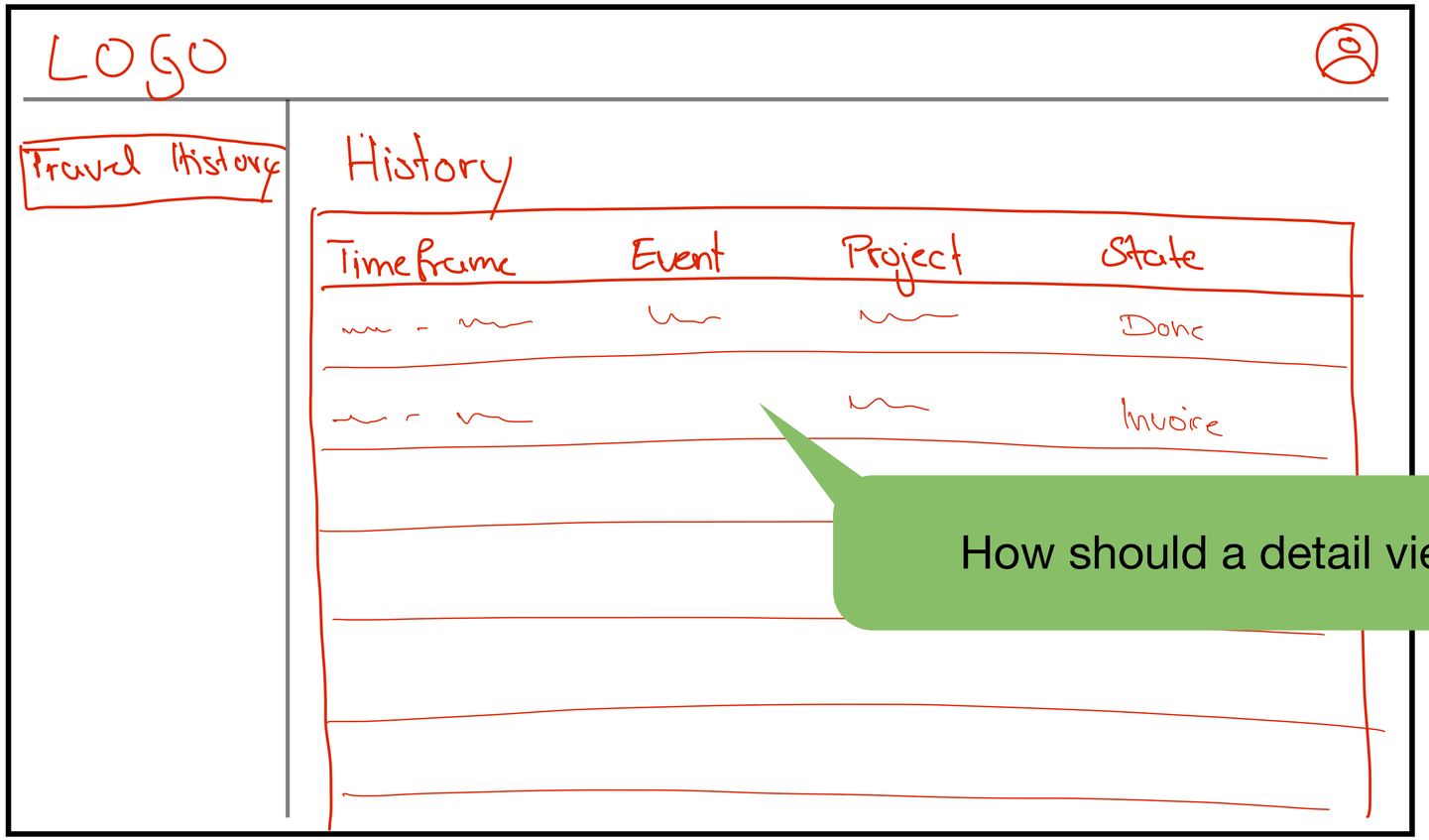
Example: Travel Booking System



Designing a Top Level Architecture

1. Pick a User Story
2. Look at participating objects in the AOM
3. Group related objects to one system
4. Model the dataflow
5. Specify the required Services
6. Adapt to comply with Design Goals

Example: Travel Booking System



Break

Matthias Linhuber

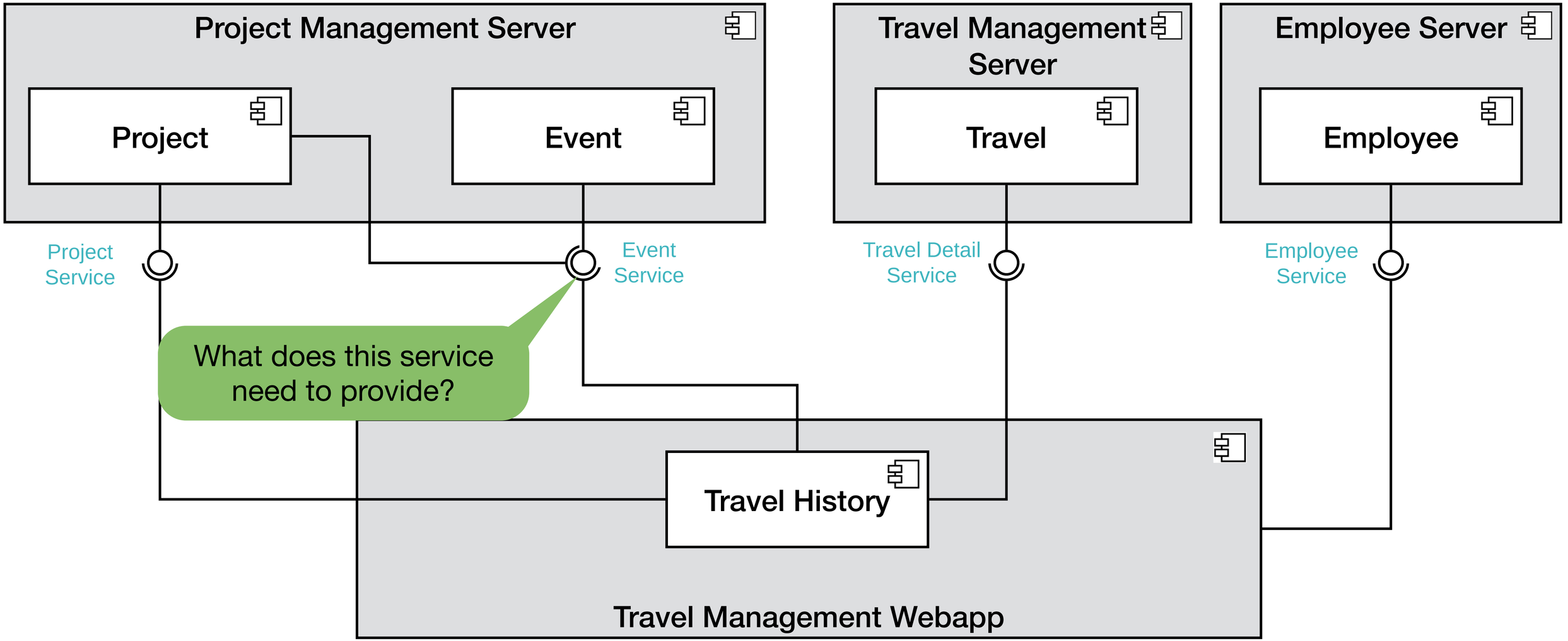
matthias@linhuber.org mtze.me github.com/Mtze

Service Based Architectures

Service Based Architectures

- The terms **Service** and **API** are often confused
- **Subsystem Service:**
  - Specifies the interaction and information flow from and to subsystem boundaries, but not inside the subsystem
  - Set of fully typed operations
- **Application programming interface** (API)
  - The API is the specification of the subsystem interface in a specific programming language / technology
  - REST / gRPC / …

Example: Travel Booking System



Speccing a Service

- What data do we expect from the service?
- What data do we need to provide to process the request?
- In which from do we need the data?
  - Single Entry?
  - Set / List?
- Who should be able to use the service?
- Request-Response or do we need a Subscription option?

From Service to API

- An API is a **set of rules that allows software entities to communicate**
- Seeds: **Services** in a component diagram
- APIs need
  - **specifications**
  - **shared understanding** of the use case
- API Design Technologies / Paradigms
  - REST: Representational State Transfer
  - GraphQL: Graph based query language
  - gRPC: Remote Procedure Calls
  - WS: WebSockets
  - MQTT: Pub-Sub Mechanisms
  - ...

Sometimes also called "Contract"

From Service to API

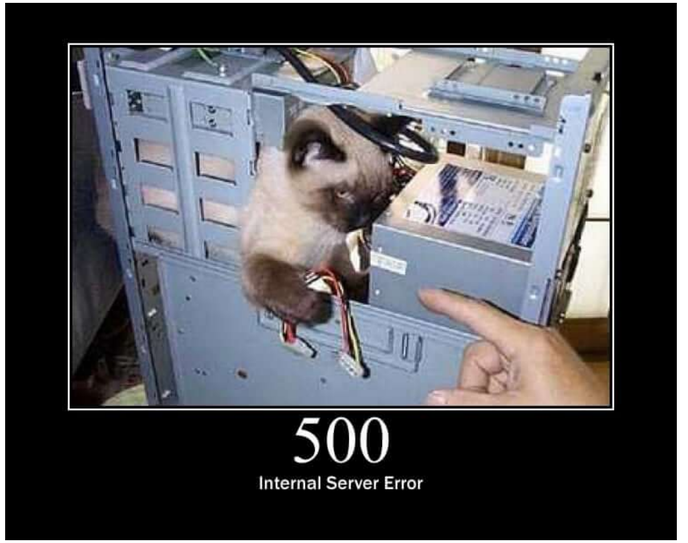
API Type	Protocol	Data Format	Standardization	Request/ Response Model	Statelessness	Security	Real-Time Communication
REST	HTTP	JSON, XML	HTTP methods, Status codes, (OpenAPI)	Request-Response	Stateless	HTTPS Token-based	Polling Long-Polling
GraphQL	HTTP, more	JSON	Schema, Types	Request-Response	Stateless	HTTPS Token-based	Subscriptions Polling
gRPC	HTTP/2	Protocol Buffers	Protocol Buffers, HTTP/2	Request-Response, Streaming	Stateless	HTTPS Token-based	Streaming Polling
WebSockets	TCP	Any	Protocol	Full-Duplex	Stateful	WSS, Custom	Native

REST

- REST is an **architectural style** that defines a set of constraints to be used when designing Client-Server Architectures
- Based on API **routes** `/api/v1/path/to/resource`
- **Stateless**:
  - Every API call **must contain all the information** needed to process the request.
  - The server does not store anything about the client state between API calls.
- Standards for
  - **access methods** (HTTP Methods)
  - **resource identifiers** (URIs / Routes)
  - **resource representations** (JSON or XML)

REST: HTTP Methods

- HTTP Methods:
  - **GET:** Retrieve information about a resource.
  - **POST:** Create a new resource.
  - **PUT:** Update an existing resource.
  - **DELETE:** Remove a resource.
  - (**PATCH:** Partially update an existing resource.)
- HTTP Codes
  - **1xx** Informational Responses: “Just management things”
  - **2xx** Success: “You’re good”
  - **3xx** Redirection: “Go away”
  - **4xx** Client Errors: “You screwed up”
  - **5xx** Server Errors: “I screwed up”

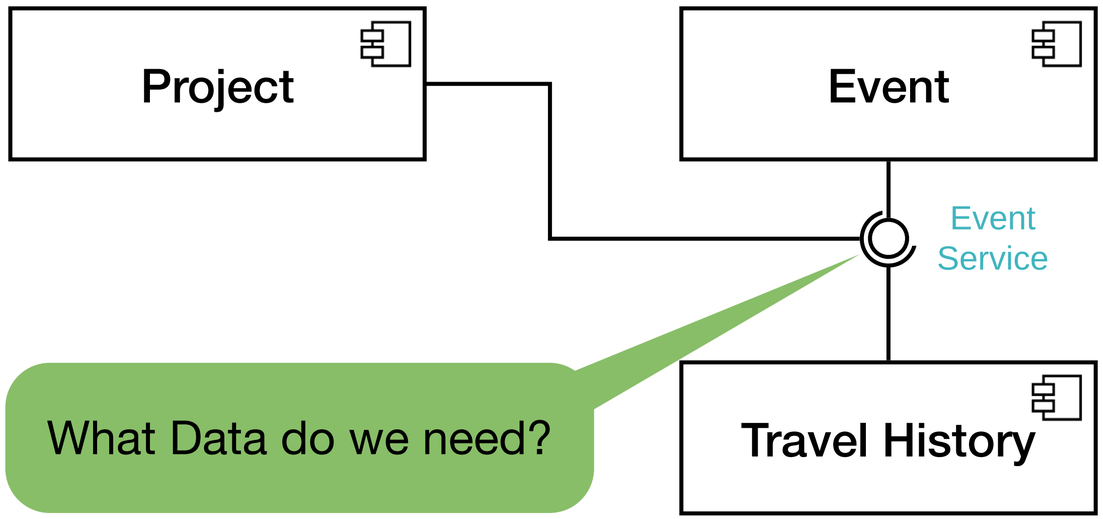


REST API Specs

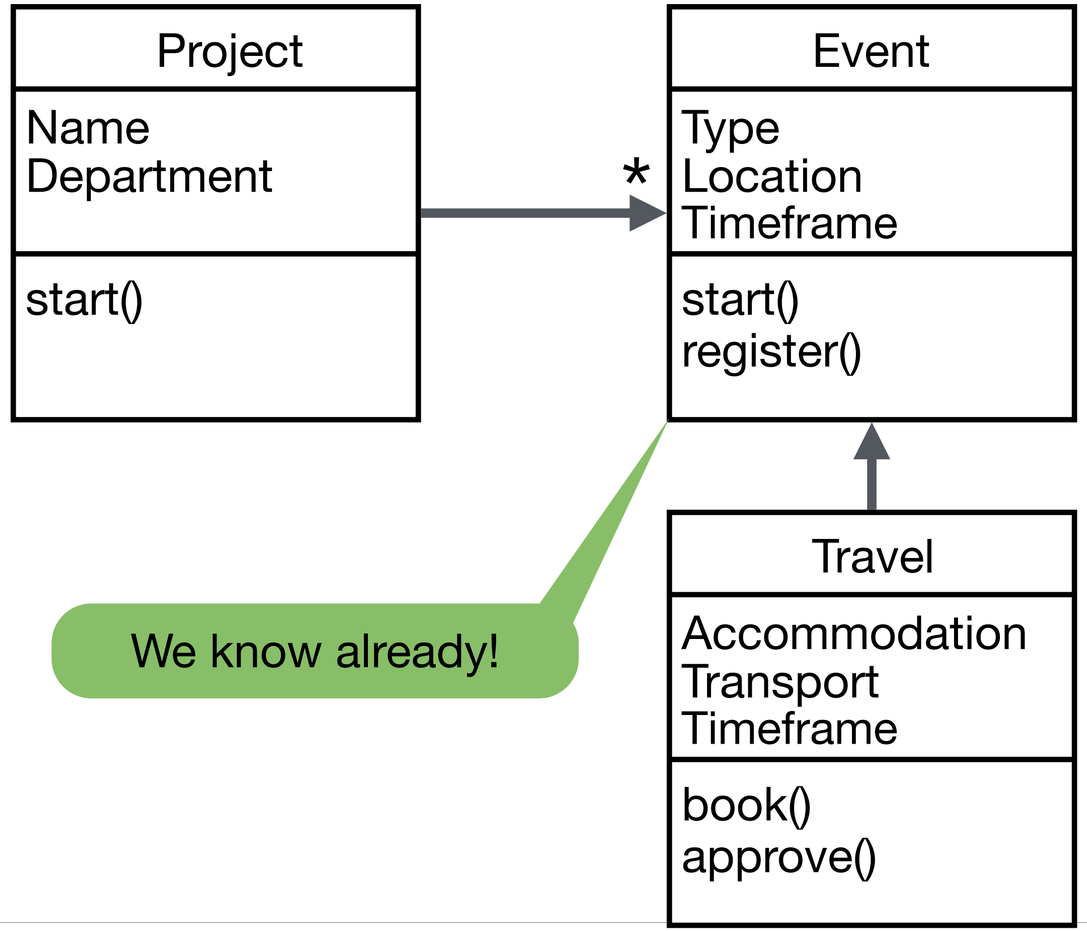
- **Language agnostic** standard to document/specify RESTful APIs in a **machine and human readable** way
- Benefits
  - Automatic API documentation (Code -> OpenAPI Spec)
  - Code Generation (OpenAPI Spec -> Code)
  - Interactive Documentation (API Browser)
  - API Validation
  - Enforces Consistency
- Specifies **Types**, **Routes**, and **Methods**

Example: Designing a REST API

Subsystem Decomposition

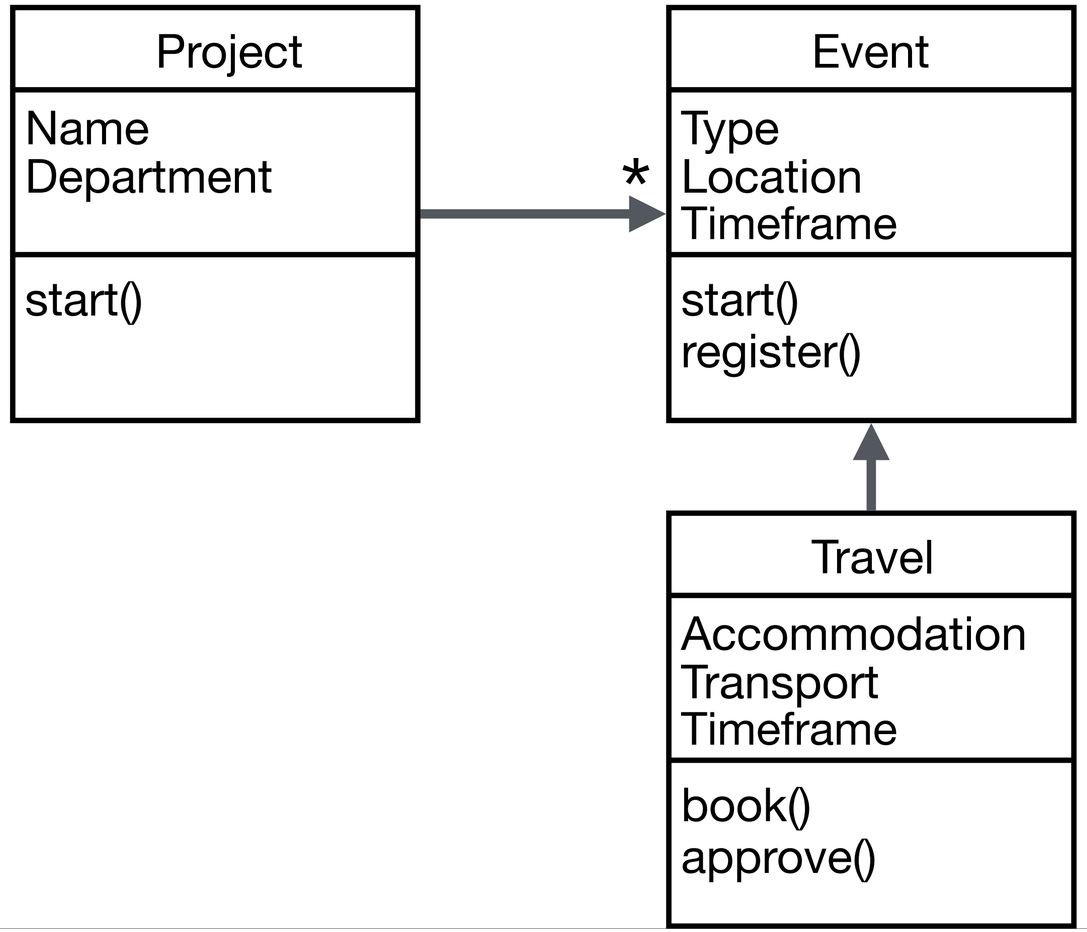


Analysis Object Model

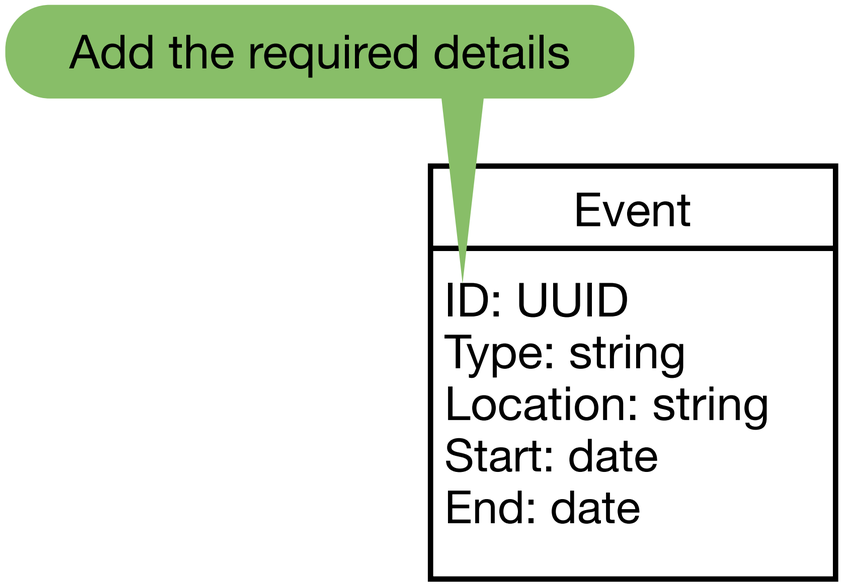


Example: Designing a REST API

Analysis Object Model

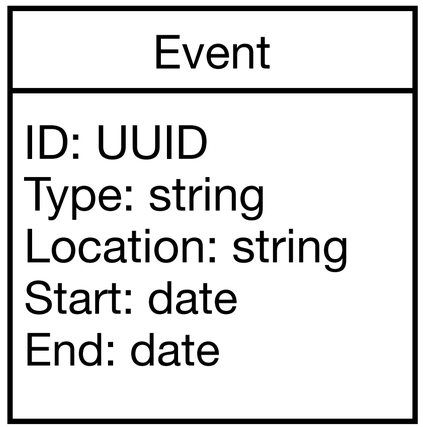


Data Model



Example: Designing a REST API

Data Model



Open API Spec

```yaml
Event:
      type: object
      properties:
        id:
          type: string
          format: uuid
          description: Unique identifier for the Event.
        startTime:
          type: string
          format: date-time
          description: Event start time.
        endTime:
          type: string
          format: date-time
          description: Event end time.
        type:
          type: string
          format: string
          description: The type of the Event - Possible options tbd
```

Example: Designing a REST API

Open API Spec

```yaml
Event:
     type: object
     properties:
       id:
         type: string
         format: uuid
         description: Unique identifier for the Event.
       startTime:
         type: string
         format: date-time
         description: Event start time.
       endTime:
         type: string
         format: date-time
         description: Event end time.
       type:
         type: string
         format: string
         description: The type of the Event - Possible options tbd
```

Code

Client: Swift

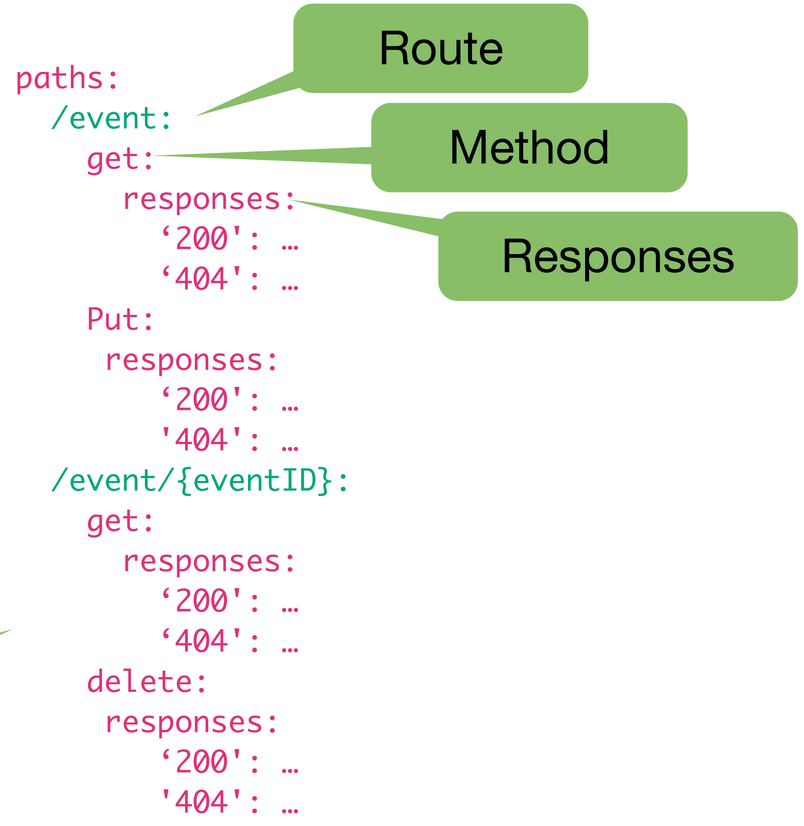
```swift
struct Event: Codable {
    let id: UUID
    let startTime: Date
    let endTime: Date
    let type: String
}
```

Server: Go

```go
type Event struct {
   ID         string    `json:"id"`
   StartTime  time.Time `json:"startTime"`
   EndTime    time.Time `json:"endTime"`
   type       string    `json:"type"`
}
```

Example: Designing a REST API

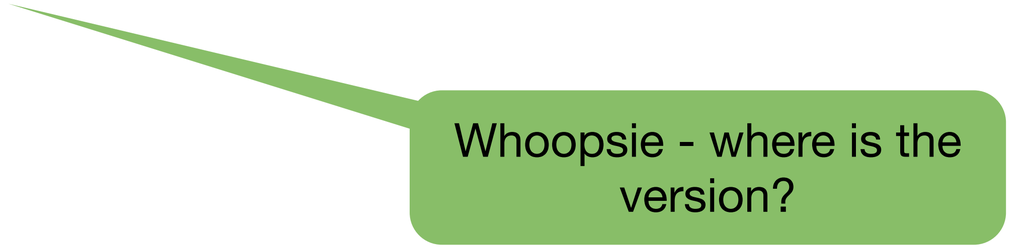
- We have the types now
- Next Step:
  - Spec **Routes** to implement the service
  - Specify the **Methods** required
  - Specify **Responses**



Do you notice a problem with our API Spec?

Example: Travel Booking System

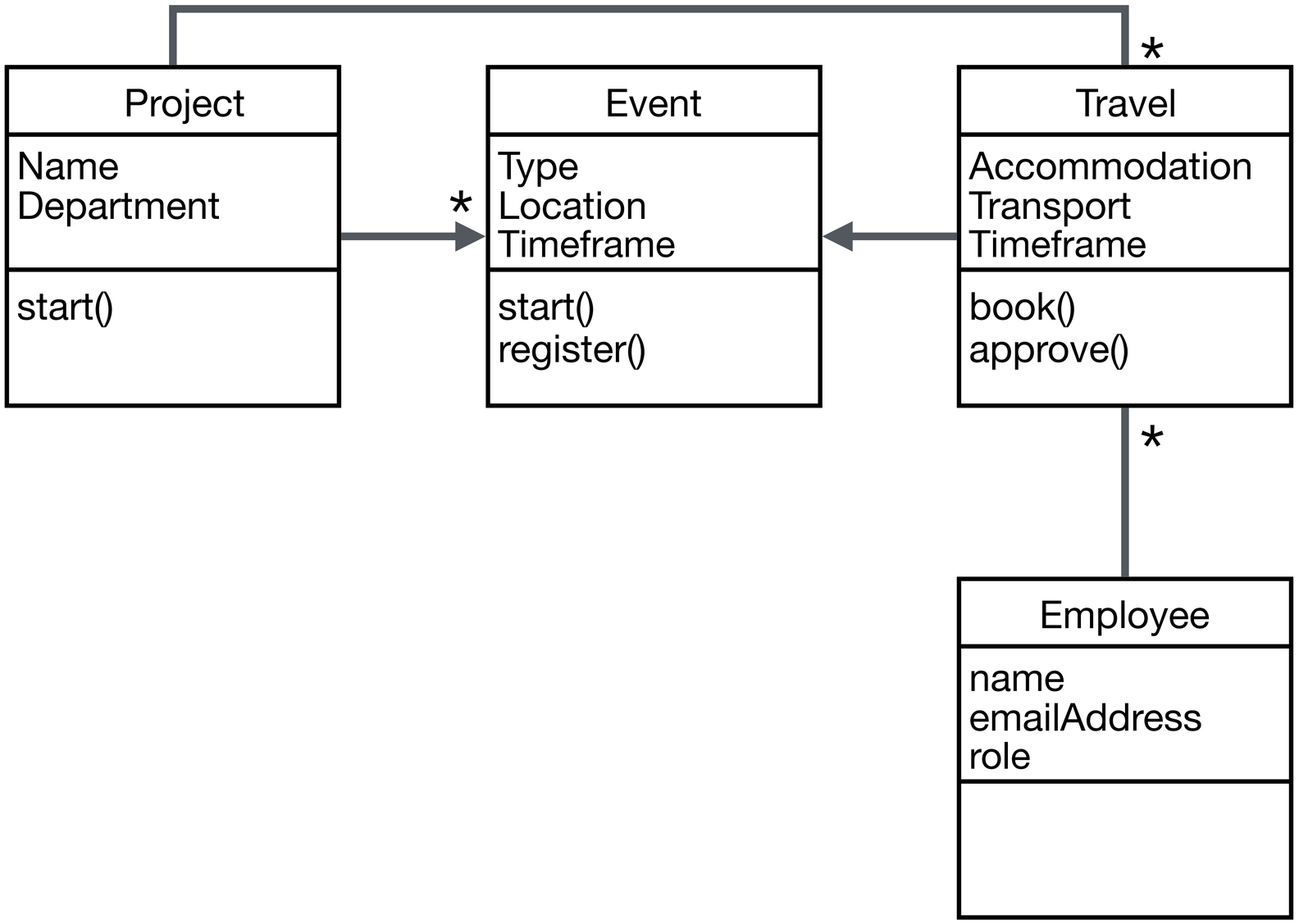
- **Design Goal**
  - The system should be implemented as micro service architecture to allow independent deployability of software components
  - All system APIs have to be versioned

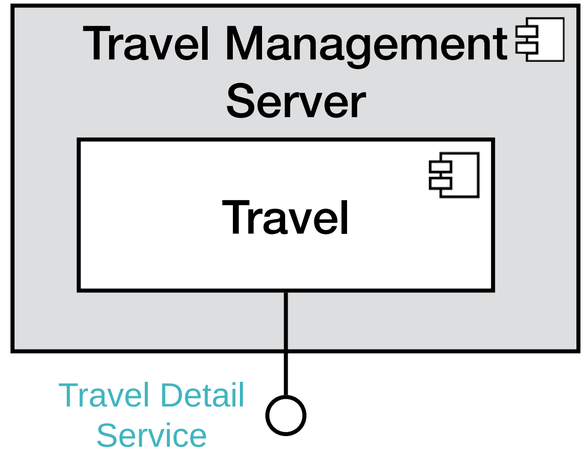


Task - Design the Travel Service

- Follow the outlined steps to spec the Travel service

Example: Travel Booking System





Data Management

Data Management

- **Persisting data** is an essential part of a product
- Questions to answer
  - **Which data** has to be stored? (Type, Amount, Required Latency, …)
  - **Where** do we need to store it? (Device, Server, External Services, …)
  - Which **storage solutions** are available? (Swift Data, SQL Database, NoSQL Database, Key-Value Store, S3, Filesystem, …)
  - Do we need **caching**? (Caching on device, caching on the server, CDN, …)

Conclusion

Recap

- Now you
  - Understand the activities to build an Architecture
  - Understand the concepts of **coupling** and **cohesion**
  - Understand the importance of **Design Goals**
  - Create a **Top Level Architecture**
  - Create a **Subsystem Decomposition**
  - Create an **API Design in a Service Based Architecture**

Where To go from here

- **Deployment** (UML Deployment Diagram, Docker, Kubernetes, …)
- **Access Control** (OIDC, Cookies, JWT, …)
- **UI/UX design** (Figma, Nielsen Usability Heuristics, HIG, SUS, …)
- **Data Management** (Databases, Encryption, …)
- **Testing** (Unit-, Integration-, User-Testing, …)
- **Monitoring** (Observability, System Metrics, Profiling, Tracing, …)

Software Architecture

A Software Architecture is never right, it is just not wrong yet!

You don’t need to be the perfect Architect, just be a tiny bit better than yesterday - everyday!

Software Architecture 101

From Requirements to a Software Product

Matthias Linhuber

matthias@linhuber.org mtze.me github.com/Mtze

References

- Software Architecture in Practice (Bass et al.)
- Designing Software Architectures (Klein et al.)
- Just Enough Software Architecture (Fairbanks)
- IEEE 42010
- Object-oriented software engineering: using UML, patterns, and Java (Bruegge, Dutoit)
- Introduction to Software Engineering (Prof Krusche)