SOFTWARE INFRASTRUCTURE

Adam Fuzum Tewelde

SHIKORINA DATING APP

FEATURES

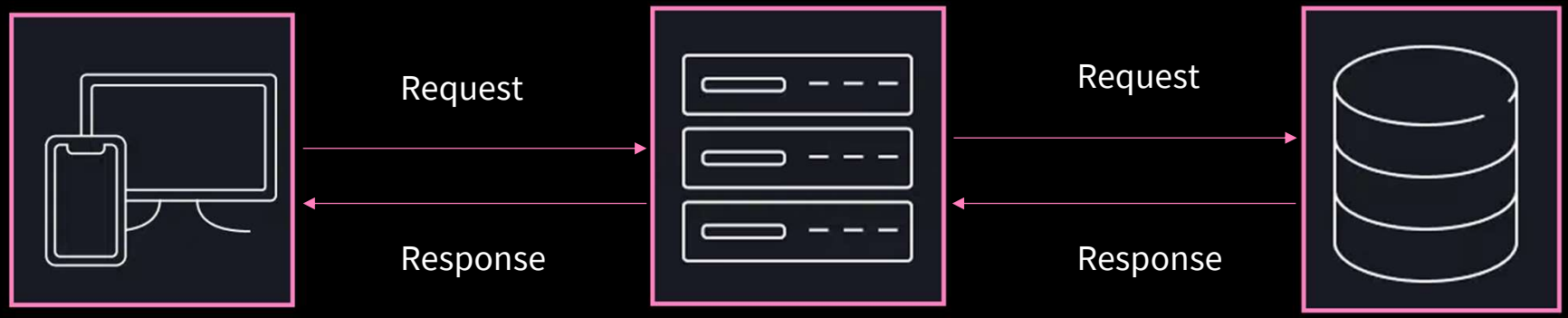
- REGISTER USERS
- USER PROFILE CUSTOMIZATION
- USER PROFILE SURFING
- USERS COMMUNICATION
- USERS ALERTING

TIERS OF THE SOFTWARE

PRESENTATION TIER
FRONTEND
USER INTERFACE

APPLICATOIN TIER
BACKEND
BUSINESS LOGIC

DATA TIER
DATA WAREHOUSE



PRESENTATION TIER

- used to interact with business logic by the end user [users]
- served from the backend- processed in the users- device [pc, self phone, etc]-
- usually built by css, html and js, some devices use c#, java
- popular frameworks include React, Angular, Vue.js, Flutter, and Xamarin**

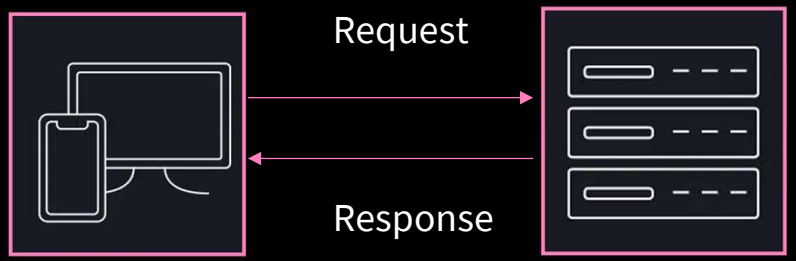
APPLICATOIN TIER

- Responsible for streaming presentation tier contents
- Responsible for interacting with the data tier and maintaining tastefulness off the software
- It is core of the software where the computation is done

DATA TIER

- Responsible for string state and application data
- Responsible for quiring data

PRESENTATIN TIER VS APPLICATION TIER COMMUNICATION



COMMUNICATION MODELS

1. REST [HTTP]
2. SOAP [UDP]

COMMUNICATION DATA TYPES

1. JSON
2. HTML
3. XML
4. TEXT
5. IMAGE
6. VIDEO

REST API MODEL

PRESENTATION TIER
CLEINT

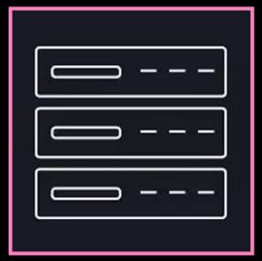
GET | POST | UPDATE | DELETE

[HTTP REQUEST]

[HTTP RESONSE]

DATA [JSON | HTML | XML | ETC…

STATUS [200 | 201 | ……..



APPLICATION TIER
SERVER

REST MODEL REQUEST RESPONSE DIAGRAM

```
Client -----> [GET Request] -----> Server
                    URL: /v1/users
                    Query Params: active=true
                    Headers: Authorization: Bearer abc123

Server -----> [Response] -------> Client
                    Status Code: 200 OK
                    Body: [
                      {"id": 1, "name": "John"},
                      {"id": 2, "name": "Jane"}
                    ]
```

REST API MODEL REQUEST COMPONENTS

1. HTTP Method

- **GET**: Retrieve data (read-only).
- **POST**: Create a new resource.
- **PUT**: Update an existing resource (replace entire resource).
- **PATCH**: Update partial fields of a resource.
- **DELETE**: Remove a resource.

2. URLRL (Uniform Resource Locator)

The URL specifies the location of the resource on the server. It is composed of:

•**Base URL**: The domain or server address (e.g., https://api.example.com).
•**Endpoint**: The specific path to the resource (e.g., /v1/users).

3. Headers
Headers provide additional information about the request, such as authentication, content type, and caching.

Common Headers:

•Authorization: Used for authentication (e.g., Bearer <token>).
•Content-Type: Specifies the format of the data being sent (e.g., application/json).
•Accept: Specifies the format the client expects in the response (e.g., application/json).
•User-Agent: Describes the client making the request (e.g., PostmanRuntime/7.29.0).

Example:

REST API MODEL REQUEST COMPONENTS

1. **HTTP Method**

- **GET**: Retrieve data (read-only).
- **POST**: Create a new resource.
- **PUT**: Update an existing resource (replace entire resource).
- **PATCH**: Update partial fields of a resource.
- **DELETE**: Remove a resource.

1. **URL (Universal resource allocator)**

The URL specifies the location of the resource on the server. It is composed of:

- **Base URL**: The domain or server address (e.g., https://api.example.com).
- **Endpoint**: The specific path to the resource (e.g., /v1/users).

REST API MODEL REQUEST COMPONENTS

3. Headers

Headers provide additional information about the request, such as authentication, content type, and caching.
Common Headers:

- Authorization: Used for authentication (e.g., Bearer <token>).
- Content-Type: Specifies the format of the data being sent (e.g., application/json).
- Accept: Specifies the format the client expects in the response (e.g., application/json).
- User-Agent: Describes the client making the request (e.g., PostmanRuntime/7.29.0).

4. Path Variable

Path variables are placeholders within the URL that are replaced with actual values at runtime. e.g /users/{id}

5. Query Parameters | Path Arguements
Query parameters are key-value pairs appended to the URL to filter or customize the request. They follow a ? and are separated by &.
e.g. https://api.example.com/v1/users?role=admin&active=true

6. Request Body | Request Payload

The request body contains the data being sent to the server (used in methods like **POST**, **PUT**, or **PATCH**).
It is typically in **JSON** or **XML** format.

REST REQUESTS EXAMPLE

```http
POST https://api.example.com/v1/users
```

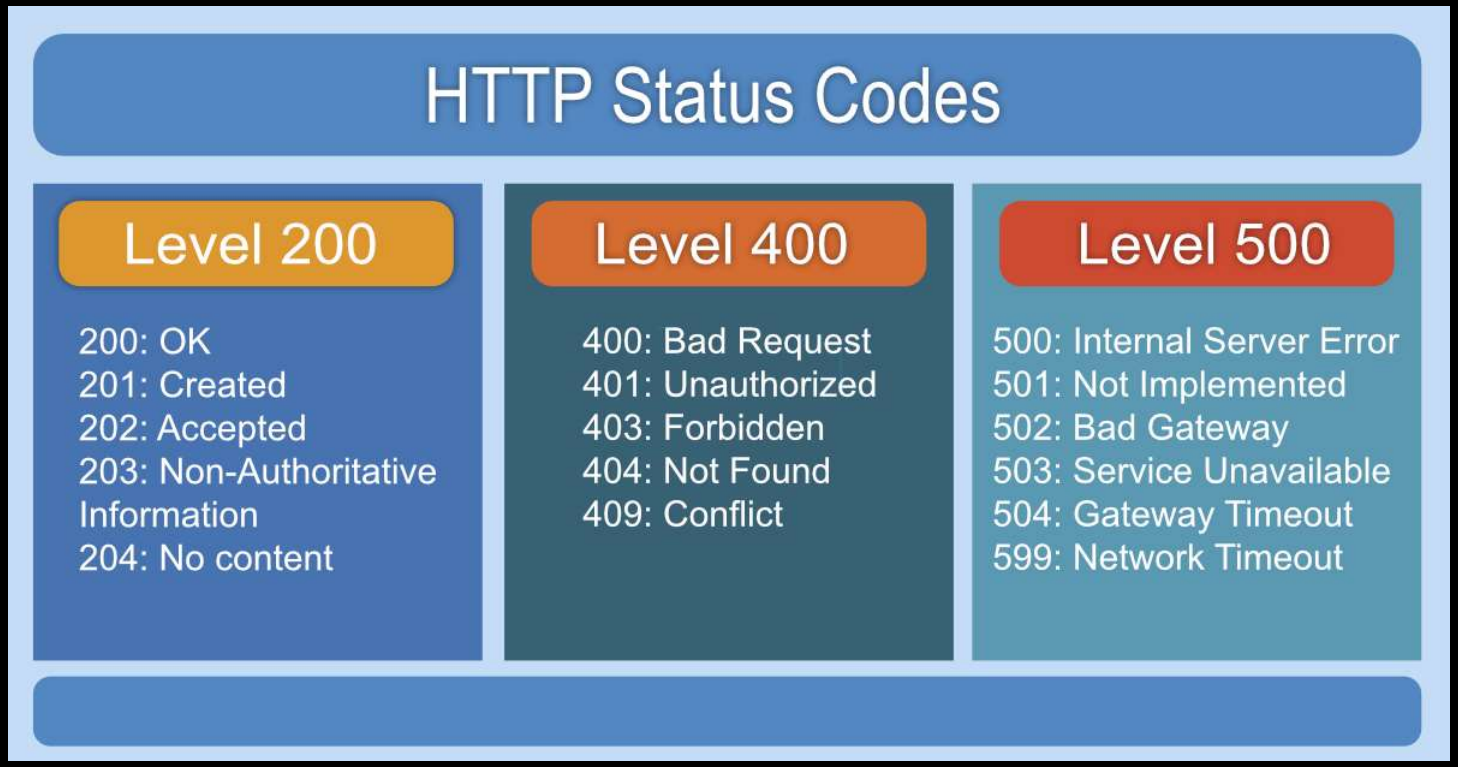
Headers:

```json
{
  "Authorization": "Bearer abc123",
  "Content-Type": "application/json",
  "Accept": "application/json"
}
```

Request Body (JSON):

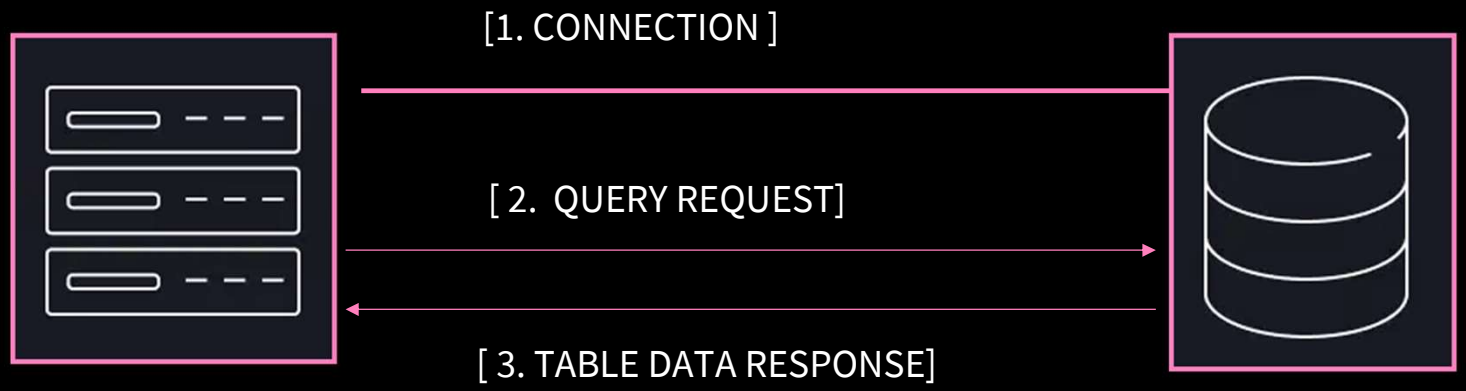
```json
{
  "name": "Jane Doe",
  "email": "janedoe@example.com",
  "role": "editor"
}
```

REST API HTTP STATUS CODES



APPLICATION TIER VS DATA TIER COMMUNICATION

The **Application Tier** communicates with the **Data Tier** using **database drivers, ORMs (Object-Relational Mappers)**, or direct **SQL queries**.



APPLICATION TIER VS DATA TIER COMMUNICATION

1. Connect

```
mysql+mysqlconnector://username:password@host:port/database_name
```

2. Query Request

```
SELECT * FROM users WHERE active = TRUE;
```

3. TABLE DATA RESPONSE

```
| id | name          | email                | active |
|----|---------------|----------------------|--------|
| 1  | Alice Johnson | alice.johnson@mail.com | TRUE   |
| 2  | Bob Smith     | bob.smith@mail.com     | TRUE   |
```

ORM [OBJECT RELATIONAL MAPING]

is a programming technique that allows developers to interact with a database using objects instead of writing raw SQL queries. It bridges the gap between **relational databases** (tables) and **object-oriented programming languages**.

In simpler terms, an **ORM maps database tables to Python classes** and rows of data to objects, allowing you to work with your database in an object-oriented way.

Why Use an ORM?

1. **Abstraction**: No need to write raw SQL queries.
2. **Code Readability**: Work with Python classes and methods instead of SQL syntax.
3. **Portability**: Makes switching between databases easier (e.g., from SQLite to MySQL).
4. **Consistency**: Ensures consistency between your database schema and application models.
5. **Ease of Use**: Simplifies complex queries, joins, and relationships between tables.

How Does ORM Work?

1.Mapping Tables to Classes:

1. A table in the database corresponds to a Python class.
2. Columns in the table map to attributes of the class.

2.Mapping Rows to Objects:

1. Each row in the table corresponds to an instance of the Python class.
2. The ORM translates between Python objects and SQL rows.

3.Performing Queries:

1. You use Python methods to query the database.
2. The ORM generates the corresponding SQL behind the scenes.

- **Popular ORM Libraries in Python**

1. SQLAlchemy
2. Django ORM
3. Peewee
4. Tortoise-ORM

Key Components of ORM

1. Model: Classes in your code that represent tables in the database.

```
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    active = Column(Boolean)
```

Key Components of ORM

2. Session: acts as a temporary workspace to interact with the database.

```
session = Session()
```

Key Components of ORM

3. Queries: ORM frameworks provide a query interface to retrieve, update, or delete data.

```
users = session.query(User).filter(User.active == True).all()
```

Key Components of ORM

4. Relationships: ORMs allow defining relationships between tables as attributes in the models.

```python
class Post(Base):
    __tablename__ = 'posts'
    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey('users.id'))
    user = relationship("User", back_populates="posts")
```

Without ORM [Python]

```python
import mysql.connector

connection = mysql.connector.connect(
    host="localhost",
    user="username",
    password="password",
    database="example_db"
)

cursor = connection.cursor()
cursor.execute("SELECT id, name FROM users WHERE active = TRUE")
rows = cursor.fetchall()

for row in rows:
    print(row)

connection.close()
```

With ORM

```python
from sqlalchemy import create_engine, Column, Integer, String, Boolean
from sqlalchemy.orm import declarative_base, sessionmaker

Base = declarative_base()

# Define a User model
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    active = Column(Boolean)

# Database connection
engine = create_engine("mysql+mysqlconnector://username:password@localhost/example_db")
Session = sessionmaker(bind=engine)
session = Session()

# Querying active users
active_users = session.query(User).filter(User.active == True).all()
for user in active_users:
    print(user.name)
```