Lecture 1

CSCI 6907.12 - Full Stack Application Engineering

Goal of the Course

- Learn how to **design** a system programs that come together to provide a single useful application on the web.
  - System - multiple parts
  - Single Application - one purpose
  - Web - platform

Other goals

- Learn to architect software such that it's not tied to a particular technology
- Learn to work in a team where skill sets of individual differs
- Learn the strengths and weaknesses of web as a platform

What is Web Application?

- Client-Server model of application
- Web Browser as client
- HTTP as the protocol for communication
- HTML/CSS/Javascript as the payload

Why Web Application?

- Web Browsers are everywhere!
  - No need to install client software
  - No need to re-implement same UI multiple times to support multiple platform
- Distribution is easy, you just need a URL
- Easy to maintain and update

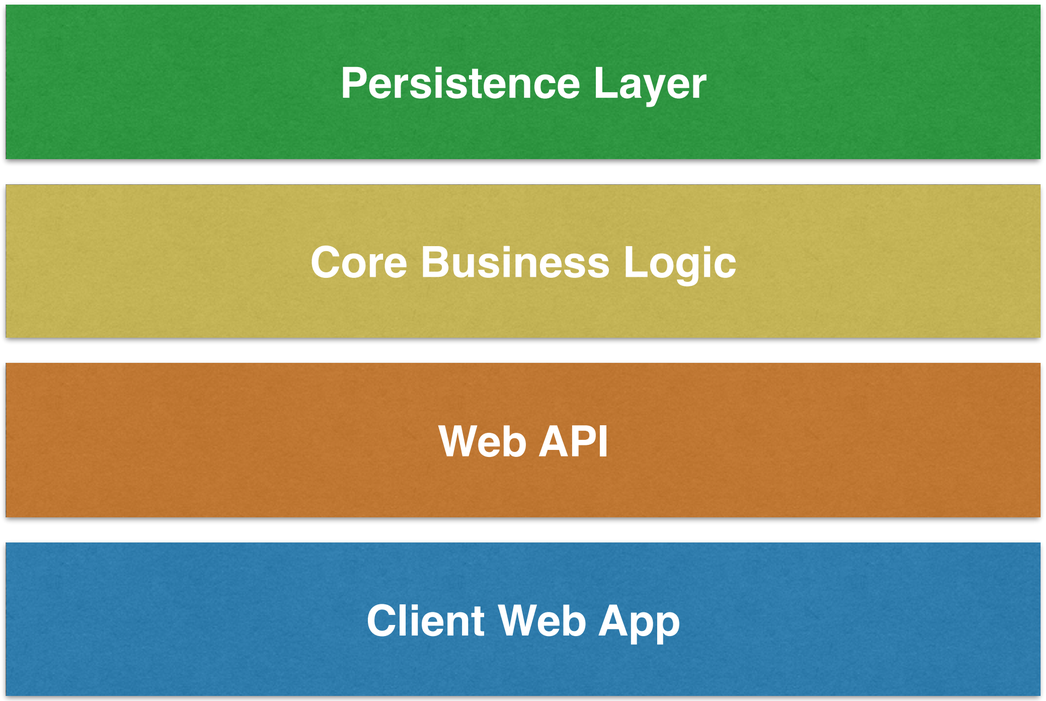
Why not Web Application?

- Overhead of network
- Overhead of browser
- Increased security risk surface

Everything is a tradeoff!

- This is not the only way of software development
- Choose your tools wisely
- Always be flexible

Big Picture



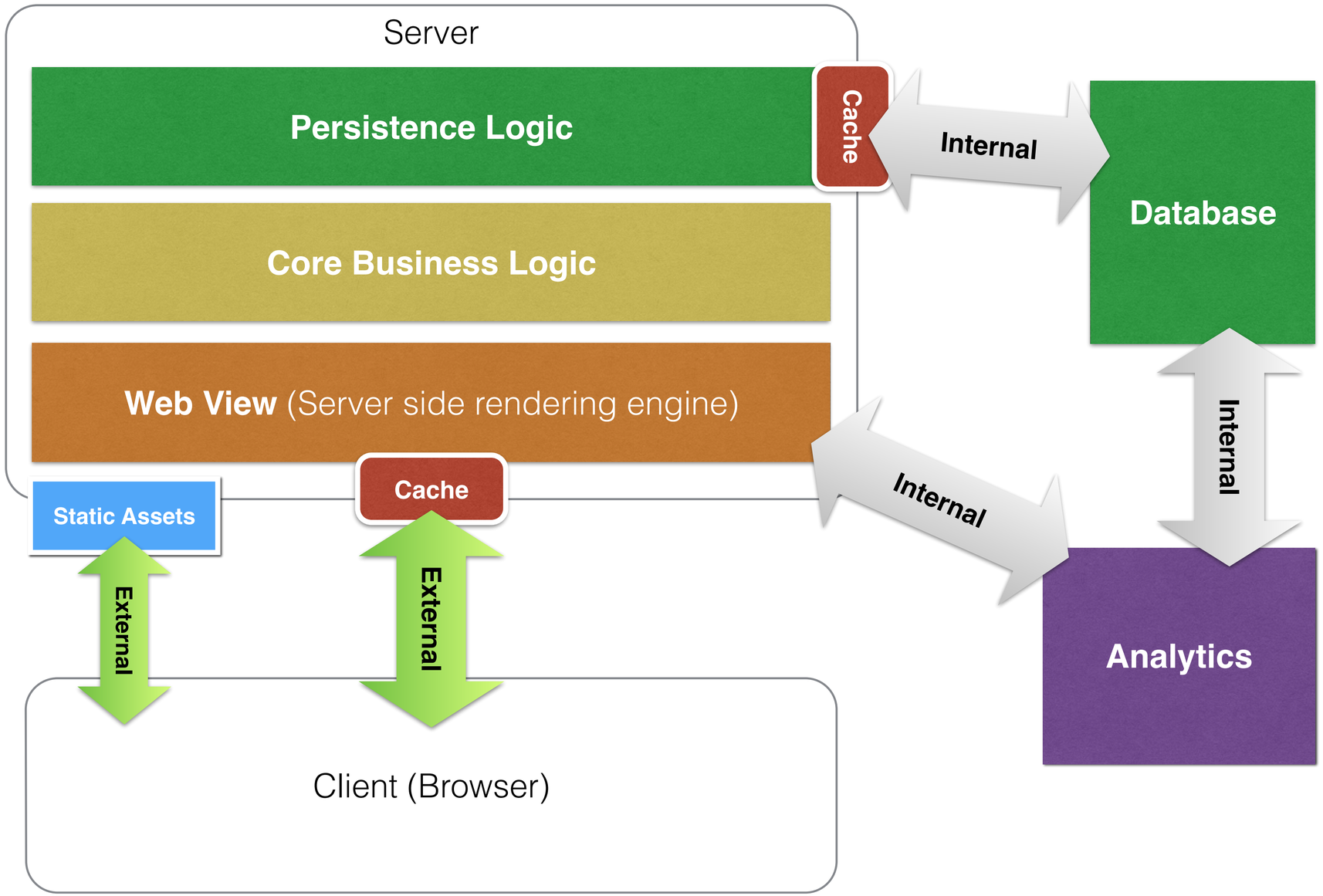
Data storage

Your core logic

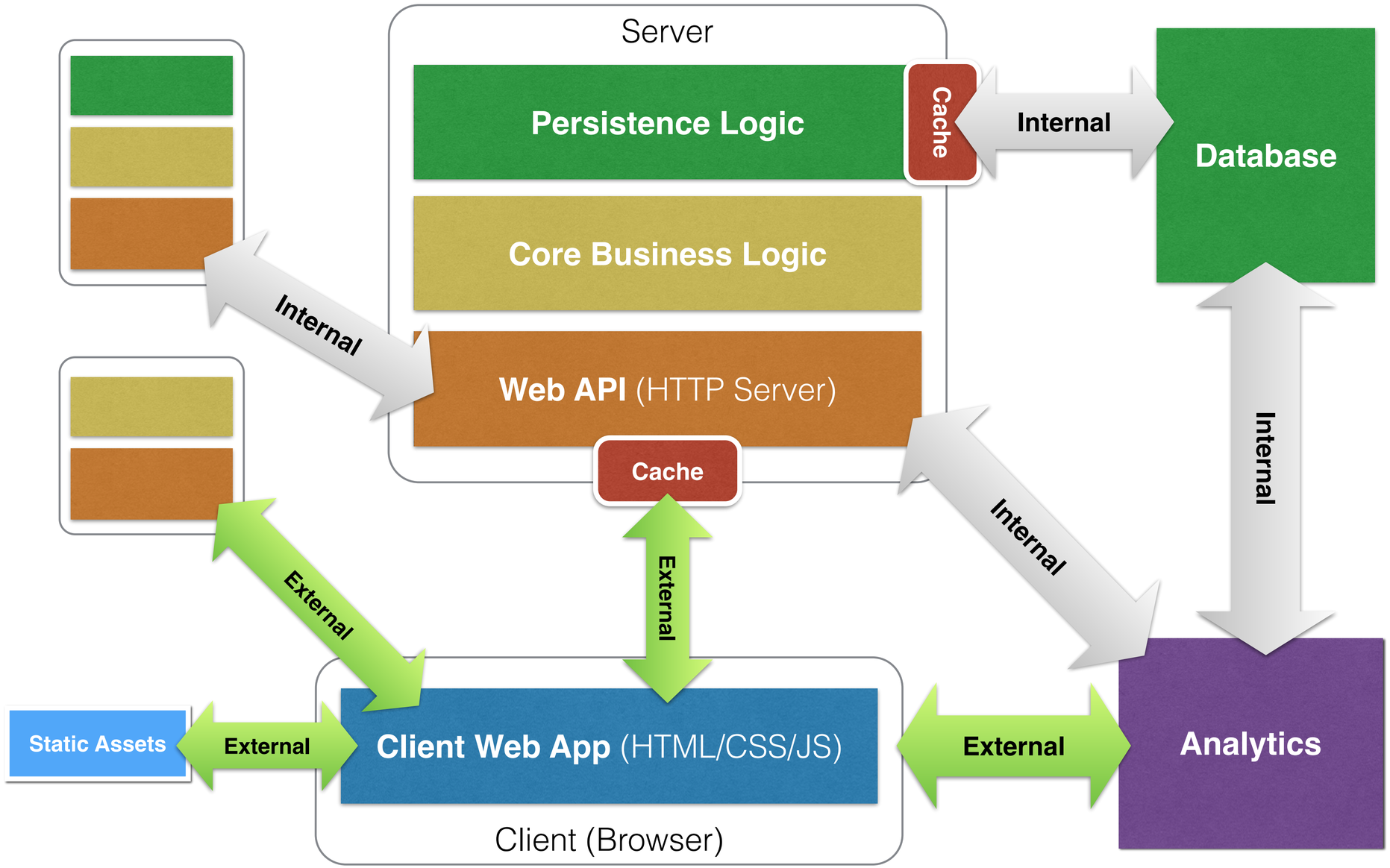
Interface to your logic

UI element

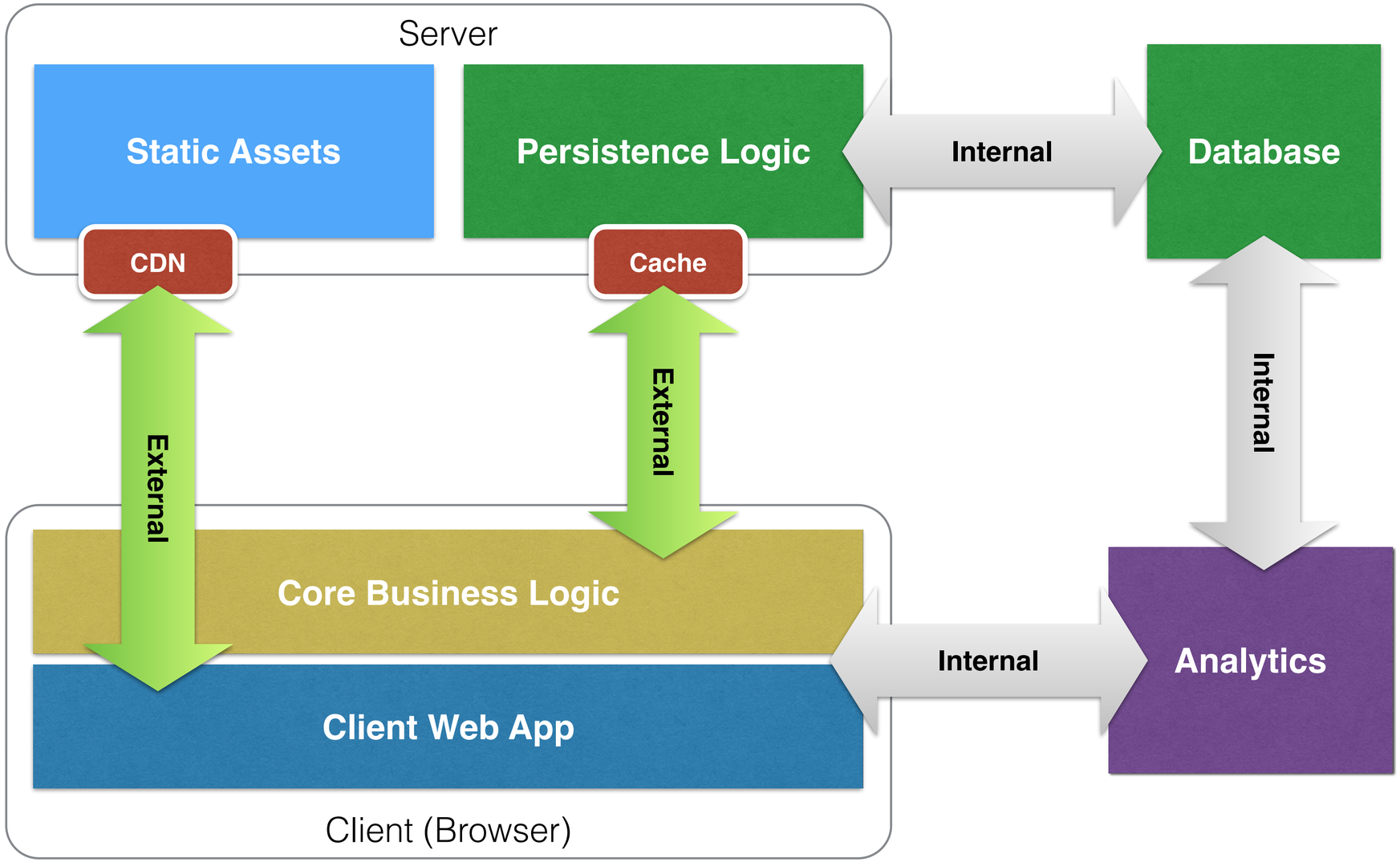
Traditional Architecture



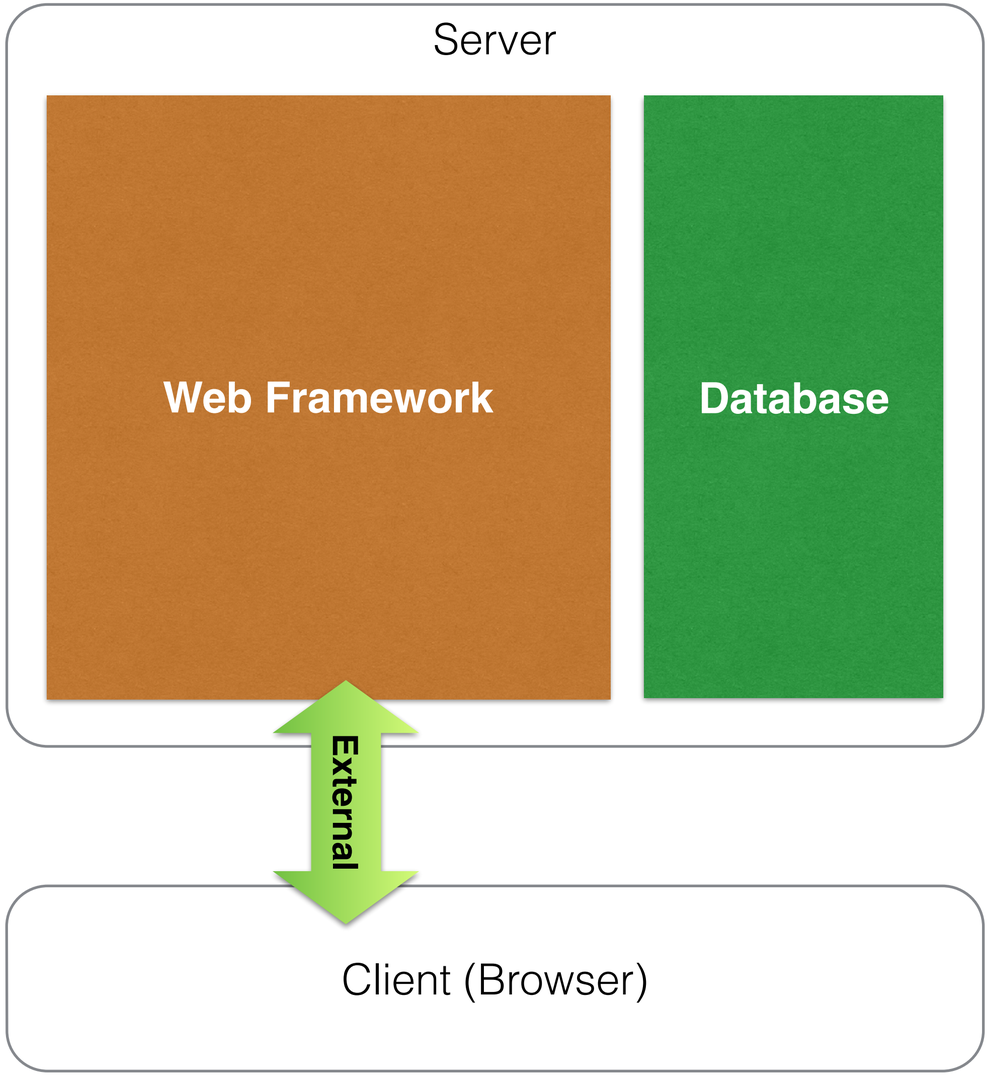
Microservice Architecture



Thick Client Architecture



Monolithic Architecture



Traditional Architecture

- **Example Technology**: LAMP Stack (Linux, Apache, MySQL, PHP)
- **Pros**:
  - Lot of resources (human and literature)
  - Relatively easy to develop due to the fact everything is in one place
- **Cons**:
  - Hard to scale
  - Hard to swap out components/technology

Microservice Architecture

- **Example Technology**: Too many to list here
- **Pros**:
  - Composed of multiple exchangeable parts
  - Easy to scale
- **Cons**:
  - Takes longer to develop due to many parts
  - You need to be polyglot to understand the entire system

Thick-client Architecture

- **Example Technology**: node.js
- **Pros**:
  - Fast development due to most of the stack is written in single place in Javascript
  - Easy to scale
- **Cons**:
  - No other choice* but Javascript for most of the code base
  - Your core algorithm is exposed to the public

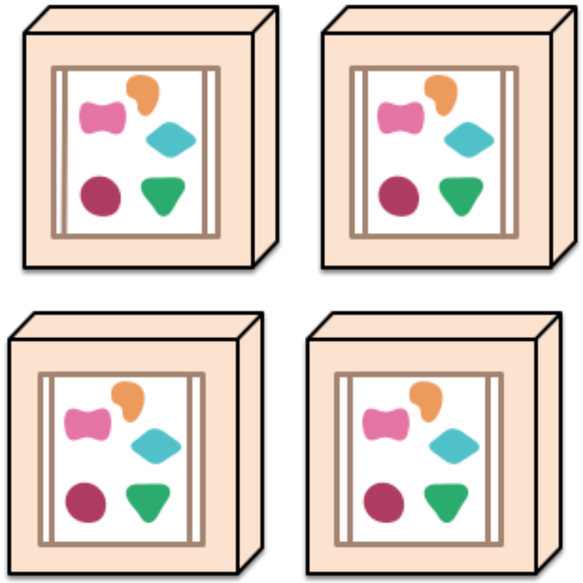
Monolithic Architecture

- **Example Technology**: Ruby On Rails, django,
- **Pros**:
  - Fast development due to most of the architectural work is done for you
- **Cons**:
  - Hard to scale
  - It's hard to steer away from the chosen technology

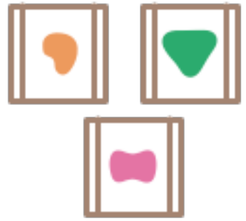
Microservice

A monolithic application puts all its functionality into a single process...

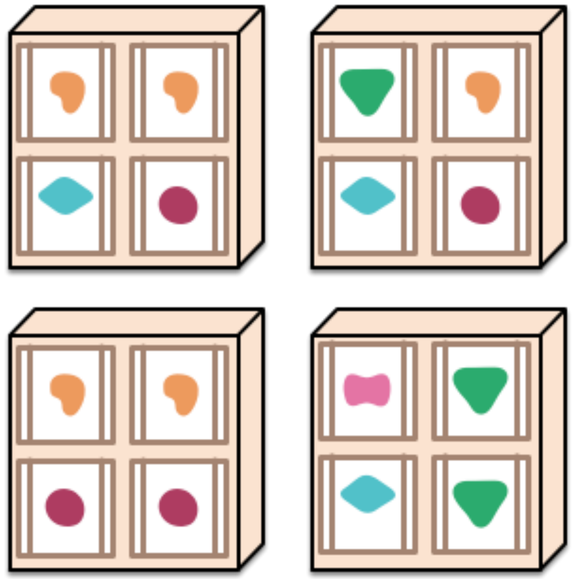
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.



It's a service

- Entire system is composed of many microservices
- Each microservice does one thing and one thing well (similar to UNIX philosophy)
- Each service organized around its capability (Stateless)
- Polyglot system
- Service oriented - all communication done through API

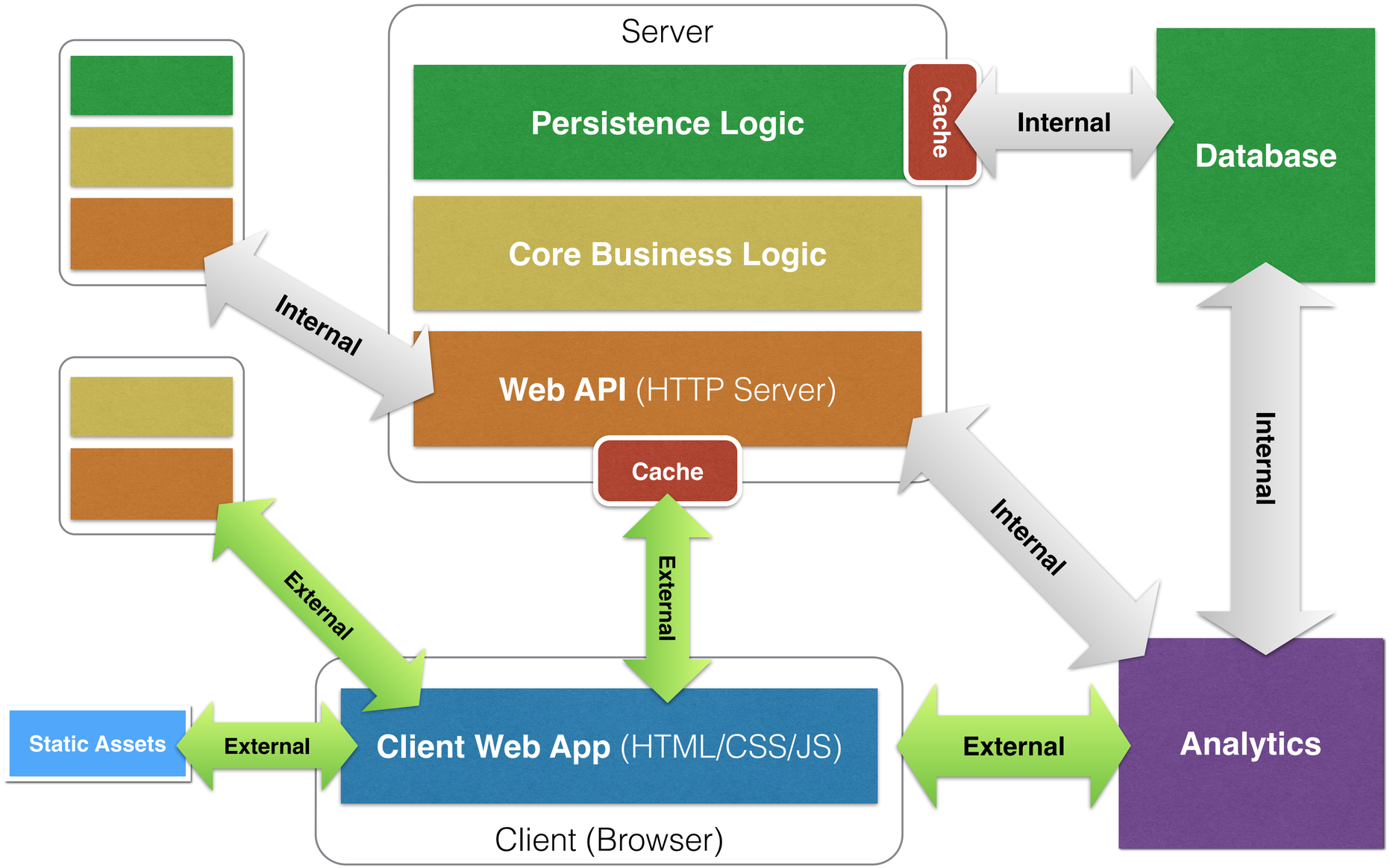
Pros

- Easily replaceable parts
- Can be developed separately by different team and different technology
- Due to symmetrical nature, each microservices can be reused for other needs
- Allows for continuous delivery

Cons

- Adds different complexity
  - Network Latency
  - API format (RESTful API tries to solve this issue)
  - Fault Tolerance (Actor model tries to solve this issue)
  - Where to draw the boundaries
- Integrated testing can be complex
- Maybe not that much different from monolithic application?

Microservice Architecture



Core Business Logic

- Brain of the application
- All computation needed to carry out the application show live here
- Should NOT contain any code regarding persistence or web
- It should be pure logic/computation
- Provides internal interface to other components

Persistence Logic

- Connects to database server
- Provides interface to persisting critical data
- The interface should be database technology agnostic
- Should be in the same language as core logic
- Often represented as ORM or DAO

Database

- Actual place where your application data gets persisted
- Your core business logic should be agnostic of its existence
- Can be relational (SQL) or document based (NoSQL)

Web API

- Provides web interface to the core business logic
- Speaks HTTP
- Should be in the same language as the core logic
- Highly recommended to use a library or a framework

Client Web App (UI)

- How the end user interact with your application
- HTML/CSS/Javascript
- Less choice here but it's getting better

Analytics

- Should work asynchronously along side the core business logic
- Lot of free and paid services out there
- Outside the scope of this class

Project Ideas