Adv Web Dev Architecture
Lecture 2

Amir Amintabar, PhD

高级Web开发架构
第2讲

阿米尔·阿明塔巴，博士

Outline

- 1 Course structure
  - Course outline
  - Announcements on the learning hub front page
  - Assessments, assignments and term project topics
- 2 What is internet software architecture all about?
- 3 The architecture we focus on: API-centric, service based
- 4 JSON as a way of data payload in API communications

大纲

- 1 课程结构
  - 课程大纲
  - 学习中心首页公告
  - 测验、作业和学期项目主题
- 2 互联网软件架构究竟是什么?
- 3 我们关注的架构：以 API 为中心、基于服务
- 4 JSON 作为 API 通信中数据载荷的方式

What is internet software/Application Architecture?

2

- **Architecture** refers to interaction/relation among components.

- In building construction: interaction/relation among walls , floors, stairs
  - Focus is on space usage, flow, beauty, cost, style

- In computer architecture: interaction/relation among CPU, Memory, Storage , I/O …
  - Focus is on speed, power usage, efficiency, computational power

- In web application: interaction/relation among UI, Database, logic, etc
  - Focus is on reliability, scalability, security, ease of implementation, modularity

什么是互联网软件/应用程序架构?

2

- **架构** 指的是各组件之间的交互/关系。

- 在建筑施工中：墙、地板、楼梯之间的交互/关系
  - 关注点在于空间利用、流动、美观、成本和风格

- 在计算机架构中：CPU、内存、存储之间的交互/关系，I/O …
  - 关注速度、功耗、效率和计算能力

- 在Web应用程序中：用户界面、数据库、逻辑等之间的交互/关系
  - 关注可靠性、可扩展性、安全性、易实现性和模块化

What are the benefits of studying architectural patterns?

- Architectural patterns help us better evaluate the efficiency , robustness, security, performance, modularity, cost of maintenance, development and extension, cost of deployment our designed application:
- **Efficiency** (e.g. I am looking for a lightweight server, what backend technology works for me),
- Cost of **maintenance** (e.g. my application is written in JavaScript and ASP .net, I need two developers. If I switch to nodejs I only need one developer good at JavaScript)
- **Development and extension** (e.g. my client wants me to add this new module to Django , but my Python developer left my company. Can I ask my nodejs guy to create a service and hook it up to Django via API? The answer is yes only if my Django application is originally developed as an API-centric application)

研究架构模式有哪些好处?

- 架构模式有助于我们更好地评估所设计应用程序的效率、健壮性、安全性、性能、模块化、维护成本、开发与扩展成本以及部署成本：
- **效率**（例如：我在寻找一个轻量级服务器，哪种后端技术适合我），
- **维护** 成本（例如：我的应用程序使用 JavaScript 和 ASP .net 编写，因此需要两名开发人员。如果我转向 nodejs，则只需要一名精通 JavaScript 的开发人员）
- **开发与扩展** （例如：客户希望我将这个新模块添加到 Django 中，但我的 Python 开发人员已离职。我可以请我的 nodejs 开发人员创建一个服务并通过 API 接入 Django 吗？只有当我的 Django 应用程序最初是作为以 API 为中心的应用开发时，答案才是肯定的）

What are the benefits of studying architectural patterns? …

- **Robustness** (e.g. is there a single point of failure in my design? How the requests' load is distributed among modules. What if the DB fails)
- **Security** (e.g. how easy I can improve the security? Should I look into every single line of my code, or there is a gateway component I originally designed which is the only portal being in touch with the outside world. That gateway is the only place I need to secure)
- **Modularity** (e.g. each module/service must carry one main functionality)
- Cost of deployment (e.g. do we need a dedicated server? If we use Ruby on Rails, perhaps we need a dedicated sever. Do we need docker?)

研究架构模式有哪些好处？ …

- **健壮性**（例如：我的设计中是否存在单点故障？请求负载如何在各模块之间分配？如果数据库发生故障怎么办）
- **安全性**（例如：我有多容易提升安全性？是否需要检查每一行代码，还是我在最初设计时就已设置了一个作为唯一对外接口的网关组件？只需保护该网关即可）
- **模块化**（例如：每个模块/服务应只承担一项主要功能）
- 部署成本（例如：我们是否需要专用服务器？如果我们使用 Ruby on Rails，可能需要专用服务器。我们需要 Docker 吗？）

There are many recognized architectural patterns and styles, among them:

有许多公认的架构模式和风格，其中包括：许多公认的架构模式和风格，其中包括：

Layered (n-Tier) Architecture

- layers = functional elements.
- Presentation layer (UI layer)
  - interaction with the end-user is handled(e.g. form page)
- Business layer (business logic)
  - controls the application's logic (e.g. decision making, routing requests)
- Application layer
  - where the core functions of the app is implemented(e.g. libraries)
- Data access layer
  - where access and retrieval of data are handled (e.g. databases)

分层（n 层）架构

- 层 = 功能元素。
- 表示层（UI 层）
  - 处理与最终用户的交互（例如表单页面）
- 业务层（业务逻辑）
  - 控制应用程序的逻辑（例如决策、路由请求）
- 应用层
  - 实现应用程序核心功能的地方（例如，库）
- 数据访问层
  - 处理数据访问和检索的地方（例如，数据库）

Service Oriented Architecture (Microservice)

- The components could be each providing a separate. All forming a collection of loosely coupled (almost independent) services. When a service reaches a specific level, it could be created in the form of a separate, stand alone app.
- We could as well split the services into smaller pieces so that each service executes a single functionality called **microservice**.
- Each service can reside in a separate machine. Be written in a different technology stack and can be up and running separately. For example the messenger app is a service to the facebook app. If facebook shuts down, it does not necessarily cause the messenger app to shut down too

SOA: Service-Oriented Architecture
MSA: Microservice Architecture slightly different but refer the same thing in this course

面向服务的架构（微服务）

• 各个组件可以各自提供独立的功能，共同构成一组松散耦合（几乎相互独立）的服务。当某个服务达到特定规模时，它可以被创建为一个独立的、可单独运行的应用程序。

• 我们也可以将服务进一步拆分为更小的部分，使得每个服务只执行单一功能，这种服务被称为**微服务**。

• 每个服务可以位于不同的机器上，使用不同的技术栈实现，并且能够独立启动和运行。例如，Messenger 应用是 Facebook 应用的一项服务，如果 Facebook 关闭，也不一定会导致 Messenger 应用随之关闭。

SOA：面向服务的架构
MSA：微服务架构 虽有细微差别，但在本课程中指的是相同的概念

Monolithic Architecture

- Opposite to microservices is monolithic architecture where in this context, means composed all in one piece.
- A monolithic architecture is the traditional unified model for the design of a software program. Often entire software resides in one machine, entire software is written in one backend technology stack.
- For example in a monolithic web application entire application is usually written in one backend technology, say Java and hosted in one machine. If you want to add something new to it you
- 1- shut down entire application
- 2- add the new feature
- 3- (may need to) recompile *everything* before deployment
  - All application logic (UI, business logic, data access) is bundled together in a single deployable unit. Simple to build and deploy but hard to scale and maintain as complexity grows.

单体架构

- 与微服务相对的是单体架构，在此上下文中，意味着所有组件集成在一个整体中。
- 单体架构是软件程序设计的传统统一模型。通常整个软件运行在一台机器上，且全部使用同一种后端技术栈编写。
- 例如，在一个单体Web应用中，整个应用通常使用一种后端技术编写，比如Java，并部署在一台机器上。如果你想向其中添加新功能，你
- 1- 关闭整个应用
- 2- 添加新功能
- 3- （可能需要）在部署前重新编译所有内容
  - 所有应用程序逻辑（用户界面、业务逻辑、数据访问）都被打包在一个可部署的单一单元中。构建和部署简单，但随着复杂性增加，难以扩展和维护。

Microservice arch. pros and cons

- Pros:
- It enables you to add new components to the system and fix any bugs without shutting everything down.(you can fix messenger app without shutting down facebook)
- One service is meant to be responsible for one function (for example, messaging, uploading files, registering users, and so on). That's why you can assign different teams to work on each service and thus speed up the development process
- Cons
- Its complicated compared with monolithic. Since each service could be hosted in a different part of the globe, the only way the services can communicate is via API. Thus you have to deal with so many API calls!
- Nonetheless, many large firms choose the Microservice architecture: Uber, Netflix, Amazon, Ebay, Sound Cloud, Groupon, realtor.com (Richmond BC)
- Breaks the system into small, independent services, each responsible for a specific business capability.Improves scalability and flexibility but increases operational complexity.

微服务架构的优缺点

- 优点：
- 它使您能够在不关闭整个系统的情况下向系统添加新组件或修复任何错误。（例如，您可以在不停止 Facebook 的情况下修复 Messenger 应用）
- 每个服务只负责一个功能（例如，消息传递、文件上传、用户注册等），因此您可以指派不同的团队分别开发各个服务，从而加快开发进程
- 缺点
- 与单体架构相比更为复杂。由于每个服务可能部署在全球不同的位置，服务之间唯一的通信方式就是通过 API。因此您必须处理大量 API 调用！
- 尽管如此，许多大型企业仍选择微服务架构：Uber、Netflix、Amazon、Ebay、Sound Cloud、Groupon、realtor.com（Richmond BC）
- 将系统拆分为多个小型、独立的服务，每个服务负责特定的业务能力。这提高了可扩展性和灵活性，但增加了运维复杂性。

Summary and more by ChatGPT (we need to verify)

- **1. Monolithic Architecture**
- **Purpose:** Quick to build and deploy for small apps or MVPs.
- Best for: prototyping, or apps that don't expect frequent independent feature updates(Q: why not suitable for this?)

- **2. Layered (n-Tier) Architecture**
- **Purpose:** Enforces separation of concerns (UI, business logic, data).
- Core banking platforms separate UI, business rules, and data persistence. This layered design ensures strict regulatory compliance, testability, and maintainability across millions of transactions.(Q: What's especial about banking platforms that we need to separate data from business ?

摘要及其他内容由 ChatGPT 生成（我们需要验证）

- **1. 单体架构**
- **目的：** 便于快速构建和部署小型应用或最小可行产品（MVP）。
- 最适合：原型设计，或不需要频繁独立更新功能的应用（问题：为何不适合此类场景？）

- **2. 分层（n 层）架构**
- **目的：** 强制实现关注点分离（用户界面、业务逻辑、数据）。
- 核心银行平台将用户界面、业务规则和数据持久化分离。这种分层设计确保了在数百万笔交易中严格遵守监管要求、可测试性和可维护性。（问题：银行平台有什么特殊之处，使得我们需要将数据与业务分离？）

- **3. Microservices Architecture**
- **Purpose:** Split complex systems into independently deployable services.
- Best for: Large-scale systems needing high scalability, agility, and independent team ownership (Netflix, Amazon Q: who else?).

- **4. Event-Driven Architecture**
- Purpose: Trigger actions asynchronously via events (publish/subscribe model).
- Best for?
- Q: Example?

- **3. 微服务架构**
- **目的：** 将复杂系统拆分为可独立部署的服务。
- 最适合：需要高可扩展性、敏捷性以及团队独立负责的大型系统（Netflix、Amazon Q：还有谁？）。

- **4. 事件驱动架构**
- 目的：通过事件（发布/订阅模型）异步触发操作。
- 最适合?
- 问：示例?

- **5. Serverless / Function-as-a-Service (FaaS)**
- **Purpose:** Run small, stateless functions in the cloud without managing servers.
- **Best for**: Apps with unpredictable traffic (the server is developed by someone else and made available to you via API calles) tasks like image processing throw sending API request to an image server, vending machine payments
- **Example**: apps you developed in term one where you had no server side scripting involved

- **6. Model-View-Controller (MVC)**
- **Purpose:** Organize code into UI (View), logic (Controller), and data (Model).
- **Best for**: Web frameworks (Django, Rails, ASP.NET MVC) — good when app logic is closely tied to views (Q: what does this mean?).
  GitHub was originally built on **Ruby on Rails (MVC)**.(true?)

- **5. 无服务器 / 函数即服务（FaaS）**
- **目的：** 在云端运行小型、无状态的函数，而无需管理服务器。
- **最适合**：流量不可预测的应用程序（服务器由他人开发，并通过API调用提供给你）例如向图像服务器发送API请求进行图像处理、自动售货机支付等任务

- **示例**：你在第一学期开发的应用程序，其中未涉及服务器端脚本

- **6. 模型-视图-控制器（MVC）**
- **目的：** 将代码组织为用户界面（View）、逻辑（Controller）和数据（Model）。
- **最适合**：Web 框架（Django、Rails、ASP.NET MVC）—— 当应用程序逻辑与视图紧密关联时效果良好（问题：这是什么意思？）。
  GitHub 最初是基于 **Ruby on Rails（MVC）** 构建的。（是真的吗？）

- **7. Model-View-ViewModel (MVVM)**
- **Purpose:** Improve UI separation, enable two-way data binding.
- Best for: Rich client-side apps with reactive UIs (Angular, Vue, React with state management).
- Real-Life Example?

- **8. Micro-Frontends**
- **Purpose:** Break large front-end apps into smaller, independently deployable parts.
- Best for: Enterprises with multiple teams contributing to a single large front-end (e-commerce sites, dashboards).
- **Real-Life Example: Spotify** uses micro-frontends to scale different parts of its web player.(true?)

- **7. 模型-视图-视图模型 (MVVM)**
- **目的：** 改进UI分离，实现双向数据绑定。
- 最适合：具有响应式UI的富客户端应用（如 Angular、Vue、结合状态管理的 React）。
- 实际例子?

- **8. 微前端**
- **目的：** 将大型前端应用拆分为更小、可独立部署的部分。

- 最适合：多个团队共同参与一个大型前端项目的企业（如电子商务网站、仪表板）。

- **实际案例：** **Spotify** 使用微前端来扩展其网页播放器的不同部分。（属实？）

- **9. Hexagonal Architecture (Ports & Adapters)**
- **Purpose:** Decouple core business logic from external systems (DBs, UI, frameworks).
- Example: Lufthansa Systems (aviation software)
- They use hexagonal architecture to decouple flight scheduling logic from databases, UI, or external APIs. This makes systems testable, reliable, and adaptable to new data sources.
- **10. CQRS (Command Query Responsibility Segregation)**
- **Purpose:** Optimize performance by separating write (commands) from read (queries).
- LinkedIn ? uses CQRS in its feed and messaging systems. Write-heavy actions (posting updates, sending messages) are handled separately from read-heavy queries (loading millions of feeds), ensuring performance at scale.

- **9. 六边形架构（端口与适配器）**
- **目的：** 将核心业务逻辑与外部系统（数据库、用户界面、框架）解耦。

- 示例：汉莎航空系统公司（航空软件）
- 他们使用六边形架构将航班调度逻辑与数据库、用户界面或外部API解耦。这使得系统更易于测试、可靠，并能适应新的数据源。

- **10. CQRS（命令查询职责分离）**
- **目的：** 通过分离写操作（命令）和读操作（查询）来优化性能。

- LinkedIn？在其动态和消息系统中使用了 CQRS。写入密集型操作（如发布更新、发送消息）与读取密集型查询（如加载数百万条动态）分开处理，从而确保大规模下的性能表现。

Can we utilize multiple architectural pattern into one products ?1/3

- Most big companies **don't stick to just one architecture** — they evolve over time and mix patterns depending on the system's needs. (examples from ChatGPT – to verify)
- **Netflix**
- **Started:** Monolithic (early DVD rental days).
- **Now:** Microservices for core streaming (recommendations, billing, video encoding).
- **Also uses:** Event-driven patterns for streaming telemetry, CQRS for separating user actions (e.g., play, pause) from analytics queries.
- Netflix shows how companies **don't stick to one architecture**. They evolve:
  Monolith → Layered → Microservices → Event-driven + CQRS as scale and complexity increase.

- **Spotify**
- **Front-end:** Micro-frontends (different teams ship UI parts independently).
- **Back-end:** Microservices (recommendations, search, music catalog).
- **Also uses:** Event-driven patterns for real-time activity feeds ("Your friend is listening to…").

能否在一个产品中使用多种架构模式？ 1/3

- 大多数大型公司 **不会局限于单一架构** —— 而是随着时间演进，并根据系统需求混合使用不同的模式。（来自 ChatGPT的例子——待验证）

Ne

- **起步阶段：** 单体架构（早期 DVD 租赁时期）。
- **如今：** 核心流媒体服务采用微服务架构（推荐、计费、视频编码）。
- **同时还使用：** 面向事件的模式用于流式遥测，CQRS 模式用于分离用户操作（如播放、暂停）与分析查询。

- Netflix 展示了企业 **不会固守单一架构**，而是持续演进：
  单体 → 分层 → 微服务 → 事件驱动 + CQRS 随着规模和复杂性的增加。

Sp

- **前端：** 微前端（不同团队独立交付用户界面部分）。
- **后端：** 微服务（推荐、搜索、音乐目录）。
- **还使用：** 事件驱动模式用于实时活动动态（"你朋友正在听……"）。

Can we utilize multiple architectural pattern into one products ? 2/3

- **Alibaba**
- **Front-end:** MVVM (Vue.js for rich, reactive UIs).
- **Back-end:** Microservices to support large-scale e-commerce operations.
- **Also uses:** Serverless (Alibaba Cloud's Function Compute) for handling burst traffic on events like Singles' Day sales.
- **LinkedIn**
- **Core:** CQRS for feeds and messaging (separating reads/writes).
- **Also uses:** Event-driven architecture for notifications, microservices for modular scaling, and hexagonal architecture patterns in some domain-driven designs.

我们能否在一个产品中使用多种架构模式？ 2/3

- **阿里巴巴**
- **前端：** MVVM（使用 Vue.js 构建丰富、响应式的用户界面）。
- **后端：** 微服务以支持大规模电子商务运营。
- **同时还使用：** 无服务器架构（阿里云函数计算），用于应对双十一购物节等事件期间的流量激增。
- **领英**
- **核心：** CQRS 用于信息流和消息传递（读写分离）。
- **还使用：** 事件驱动架构用于通知，微服务用于模块化扩展，以及在某些领域驱动设计中采用六边形架构模式。

Can we utilize multiple architectural pattern into one products ? 3/3

- **Coca-Cola (IoT / vending machines)**
- **Uses:** Serverless (AWS Lambda) for vending transactions.
- **Also uses:** Event-driven workflows for telemetry and inventory reporting, plus microservices for backend processing.

- **Lufthansa Systems**
- **Core:** Hexagonal architecture (domain-driven flight scheduling). Q: Why Hexa?
- **Also uses:** Layered architecture for ERP-like internal tools, event-driven systems for real-time flight updates.
- **Uber**
- **Started:** Monolithic Ruby on Rails app.
- **Now:** Migrated to Microservices to scale different domains (maps, payments, ride matching).
- **Also uses:** Event-driven architecture for real-time ride dispatch and pricing updates.

我们能否在一款产品中采用多种架构模式？ 3/3

- **可口可乐（物联网 / 自动售货机）**
- **使用：** 无服务器架构（AWS Lambda）处理自动售货交易。
- **还使用：** 事件驱动的工作流用于遥测和库存报告，以及用于后端处理的微服务。

- **汉莎航空系统**
- **核心：** 六边形架构（领域驱动的航班调度）。问题：为何选择六边形架构?
- **还使用：** 用于类似ERP的内部工具的分层架构，用于实时航班更新的事件驱动系统。

- **[illegible]**
- **起步于：** 单体式Ruby on Rails应用。
- **现在：** 已迁移到微服务，以扩展不同领域（地图、支付、乘车匹配）。
- **还使用：** 用于实时派单和价格更新的事件驱动架构。

3

What we study in this course

An API-Server with service based architecture

which is also RESTful

Q: it fully rests ?

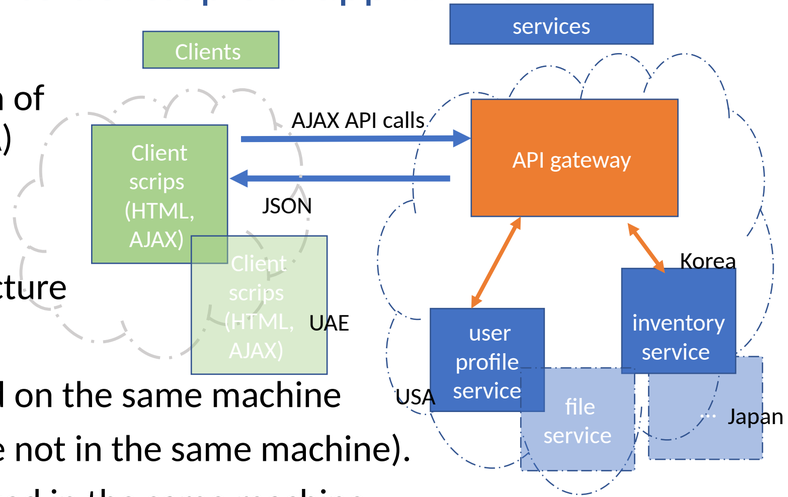
3

本课程学习内容基于服务架构的API服务器

这也是 RESTful Q:
它完全依赖吗?

19

Web architecture we use to develop our app in this course



We practice a simplified version of Microservice architecture (MSA)

We design an API centric MSA

API:

API calls follow **RESTful** architecture

Different origin:

Clint and servers are not hosted on the same machine

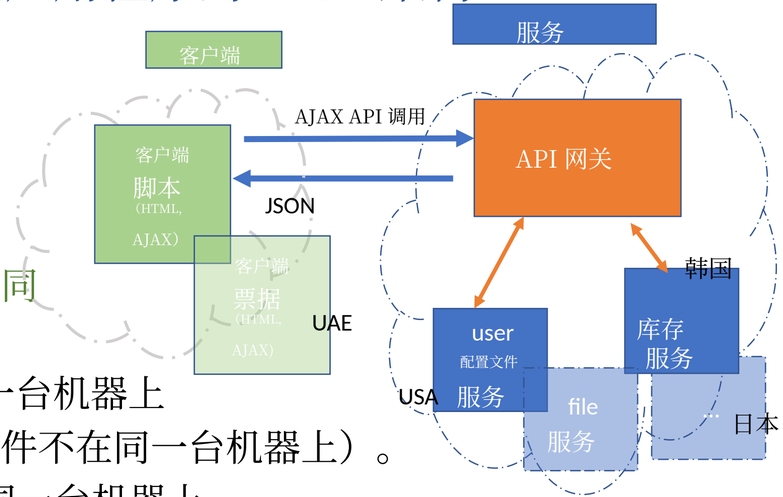
(HTML files and nodejs files are not in the same machine).

Even services might not be hosted in the same machine.

Response:

Responses coming back from services are often in **JSON** format

本课程中我们用于开发应用程序的 Web 架构



我们采用一种简化的版本微服务架构 (MSA)

我们设计以 API 为中心的 MSA

API:

API 调用遵循 **RESTful** 架构不同源：

客户端和服务端未托管在同一台机器上（HTML 文件和 Node.js 文件不在同一台机器上）。

甚至服务也可能不会托管在同一台机器上。

响应：

从服务返回的响应通常采用**JSON**格式

Learning outcome we seek from developing API centric architecture

- Design and develop programs using REST, JSON and HTTP.
  - Learning the RESTful API architecture
  - Working with **HTTP methods** (GET, POST, PUT etc)
  - Working with **JSON** as a format to pass data back and forth
- Design a simple web service.
  - A simple web service that returns result to AJAX requests like mydomain.com/API/Patients/1
- Design and develop a RESTful API by applying the best practices and REST constraints.
  - You practice this in one of your labs and in your term project
- Document and write specifications.
  - using swagger

我们希望通过开发以 API 为中心的架构所获得的学习成果

- 使用 REST、JSON 和 HTTP 设计和开发程序。
  - 学习 RESTful API 架构
  - 使用 **HTTP 方法** （GET、POST、PUT 等）
  - 使用 **JSON** 作为来回传递数据的格式
- 设计一个简单的 Web 服务。
  - 一个能像 mydomain.com/API/Patients/1 这样将结果返回给 AJAX 请求的简单 Web 服务
- 通过应用最佳实践和REST约束来设计和开发RESTful API。
  - 你将在其中一个实验和你的学期项目中进行练习。
- 编写文档并撰写规范。
  - 使用swagger

- Practice implementation of CRUD operations.
  - Review of RDBM
  - Doing **C**reate, **R**ead, **U**pdate and **D**elete operations on a database via API requests
- Implement error handling, HTTP status codes, secure API.
  - 404, 303, 200, 505, …with messages such as
  - Bad request
  - Unauthorized
  - Not found
- Apply change management and versioning of designed APIs.
  - mydomain.com/**V1**/API/Patients/1
  - mydomain.com/**V1.1**/API/Patients/1
- Secure a backend API server.
  - simple token based approach
  - Hashing using promises

- 练习实现 CRUD 操作。
  - 关系型数据库管理系统（RDBM）回顾
  - 通过 API 请求在数据库上执行**C**reate（创建）、**R**ead（读取）、**U**pdate（更新）和**D**elete（删除）操作
- 实现错误处理、HTTP 状态码、安全的 API。
  - 404，303，200，505，……以及诸如
  - 错误请求
  - 未授权
  - 未找到
- 对设计的 API 应用变更管理和版本控制。
  - mydomain.com/**V1**/API/Patients/1
  - mydomain.com/**V1.1**/API/Patients/1
- 保护后端 API 服务器的安全。
  - 基于简单令牌的方法
  - 使用承诺进行哈希

- Also
  - Working with web storage
  - Private key/ public key
  - Asynchronous programming
  - Practicing denial of service attack
  - How to design scalable applications
  - How oAuth works
  - You may need to install SSL (depends on your host provider how to do it)

- 此外
  - 使用 Web 存储
  - 私钥/公钥
  - 异步编程
  - 练习拒绝服务攻击
  - 如何设计可扩展的应用程序
  - oAuth 的工作原理
  - 您可能需要安装 SSL（具体操作取决于您的主机提供商）

Lets get started ….

让我们开始吧……

24

JSON

and

HTML Web Storage

4

JSON和

HTML Web 存储

4

Web Storage (inside your browser, not server)

- We want to store data locally within the user's browser.
- Before HTML5, application data had to be stored in cookies.
- Web storage is more secure, and large amounts of data can be stored locally (generally a couple of mega bytes, 5MB or up) .
- Note that web storage is per origin! That means per domain and protocol.
- All pages, from same origin, can store and access the same data **within same browser** on your computer of course.
- It is a client thing (in your browser's local data)
- Web storage is client-side storage, which means it resides in the local data storage of your browser

Web 存储（在您的浏览器内部，而非服务器）

- 我们希望在用户的浏览器中本地存储数据。
- 在 HTML5 之前，应用程序数据必须存储在 Cookie 中。
- Web 存储更加安全，并且可以本地存储大量数据（通常为几兆字节，5MB 或更高）。
- 请注意，Web 存储是按源（origin）划分的！这意味着按域名和协议划分。

- 当然，来自同一源的所有页面都可以在同一台计算机的 **同一浏览器** 内存储和访问相同的数据。
- 这是客户端的问题（位于浏览器的本地数据中）
- Web 存储是一种客户端存储，意味着它位于浏览器的本地数据存储中

localStorage vs sessionStorage

- There are two HTML web storage objects for storing data on the client:

- window.localStorage - stores data with **no expiration** date
- window.sessionStorage - stores data for one session (data is lost when the browser tab is closed)

- Since it is a new thing, check browser support for localStorage and sessionStorage before using web storage, :

```
if (typeof(Storage) !== "undefined") {
  // Code for sessionStorage/ localStorage.
} else {
  // Web Storage is not supported in this browser..
}
```


localStorage 与 sessionStorage

- HTML 提供了两种客户端数据存储的 Web 存储对象：

- window.localStorage — 以无过期**时间**日期的方式存储数据
- window.sessionStorage — 为单次会话存储数据（当浏览器标签页关闭时，数据将丢失）

- 由于这是较新的功能，请检查浏览器对 localStorage 和使用 Web 存储之前，sessionStorage：

```
if (typeof(Storage) !== "undefined") {
  // Code for sessionStorage/ localStorage.
} else {
  // Web Storage is not supported in this browser..
}
```

The localStorage Object

- no expiration date.
- The data will not be deleted when the browser is closed
- will be available the next day, week, or year.
- Example

```
// Store
localStorage.setItem("myKey", "Monday and Friday");
```

```
// Retrieve
localStorage.getItem("myKey");
```

- **Note:** Name/value pairs are always stored as strings. Remember to convert them to another format when needed!

localStorage 对象

- 没有过期时间。
- 当浏览器关闭时，数据不会被删除。
- 数据在第二天、下周或明年仍然可用。
- 示例

```
// Store
localStorage.setItem("myKey", "Monday and Friday");
```

```
// Retrieve
localStorage.getItem("myKey");
```

- **注意：** 名称/值对始终以字符串形式存储。需要时记得将它们转换为另一种格式！

Example

- You write stuff in a browser tab (open writer.html)
- reader.html reads the same stuff written(stored) by writer.html next day or same time in different tab

示例

- 您在浏览器标签页中编写内容（打开 writer.html）
- reader.html 可在第二天或
  不同标签页中的同一时间

The localStorage Object Example/ writer.html

This code snippet writes something onto local storage

```
<html lang="en">
<body>
    <script>
        const msg_notSupported = "Sorry web Storage is not supported!";
        const msg_key = "hidden secret";
        const msg_written="A piece of data was written in local storage for the key:"
;
        if (typeof (Storage) == "undefined") {

            document.write(msg_notSupported);
            window.stop();
        }
        localStorage.setItem(msg_key, "2021");
        document.write(msg_written+msg_key);
    </script>
</body>
```


The localStorage Object Example/ writer.html

This code snippet writes something onto local storage

```
<html lang="en">
<body>
    <script>
        const msg_notSupported = "Sorry web Storage is not supported!";
        const msg_key = "hidden secret";
        const msg_written="A piece of data was written in local storage for the key:"
;
        if (typeof (Storage) == "undefined") {

            document.write(msg_notSupported);
            window.stop();
        }
        localStorage.setItem(msg_key, "2021");
        document.write(msg_written+msg_key);
    </script>
</body>
```

30

The localStorage Object Example/ reader.html

This code snippet tries to read from local storage (remember same domain, same browser, same machine)

```
<html lang="en">
<body>
    <script>
        const msg_notSupported = "Sorry web Storage is not supported!";
        const msg_key = "hidden secret";
        const msg_read="stored data for the key ";
        if (typeof (Storage) == "undefined") {

            document.write(msg_notSupported);
            window.stop();
        }
        document.write(msg_read+msg_key+":"+localStorage.getItem(msg_key));
    </script>
</body>
</html>
```


localStorage 对象示例/ reader.html

This code snippet tries to read from local storage (remember same domain, same browser, same machine)

```
<html lang="en">
<body>
    <script>
        const msg_notSupported = "Sorry web Storage is not supported!";
        const msg_key = "hidden secret";
        const msg_read="stored data for the key ";
        if (typeof (Storage) == "undefined") {

            document.write(msg_notSupported);
            window.stop();
        }
        document.write(msg_read+msg_key+":"+localStorage.getItem(msg_key));
    </script>
</body>
</html>
```

Question!

How can I store the entire object into the local storage ?

问题!

如何将整个对象存储到本地存储中?

JSON

JavaScript Object Notation

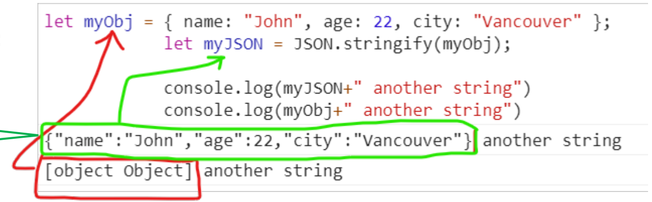
JSONJavaScript对象表示法

JSON or JavaScript Object Notation

1. You can flatten a JavaScript object into a JSON string and send it to another computer over the net.

```
let myObj = { name: "John", age: 22, city: "Vancouver" };
let myJSON = JSON.stringify(myObj);
console.log(myJSON + " another string");
console.log(myObj + " another string");
```

As you have noticed, json behaves like a flat string See the result of concatenating it with another string .



- **Q:** What is the type of myJSON here?

2. At the other end the json string can be parsed back to an object:

- `let myJSON = '{"name":"John", "age":31, "city":"New York"}';`
- `let myObj = JSON.parse(myJSON);`
- `console.log(myObj);`

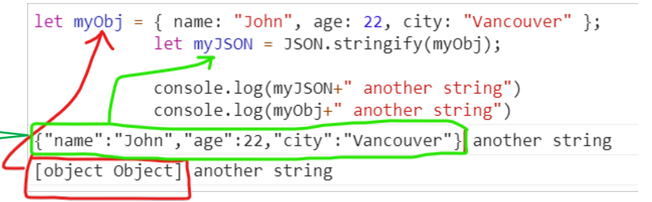
```
let myJSON = '{"name":"John", "age":31, "city":"New York"}';
let myObj = JSON.parse(myJSON);
console.log(myObj);
▼{name: "John", age: 31, city: "New York"}
    age: 31
    city: "New York"
    name: "John"
```


JSON 或 JavaScript 对象表示法

1. 你可以将一个 JavaScript 对象展平为一个 JSON 字符串，并通过网络发送到另一台计算机。

```
let myObj = { name: "John", age: 22, city: "Vancouver" };
let myJSON = JSON.stringify(myObj);
console.log(myJSON + " another string");
console.log(myObj + " another string");
```

As you have noticed, json behaves like a flat string 查看将其与另一个字符串连接后的结果。



- **Q:** What is the type of myJSON here?

2. 在另一端，可以将该 json 字符串解析回一个对象：

- `let myJSON = '{"name":"John", "age":31, "city":"New York"}';`
- `let myObj = JSON.parse(myJSON);`
- `console.log(myObj);`

```
let myJSON = '{"name":"John", "age":31, "city":"New York"}';
let myObj = JSON.parse(myJSON);
console.log(myObj);
▼{name: "John", age: 31, city: "New York"}
    age: 31
    city: "New York"
    name: "John"
```

5

More programming Tips

5

更多编程技巧

Using same event handler
for multiple buttons!
(based on JavaScript closure)

为多个按钮使用相同的事件处理程序！（基于 JavaScript 闭包）

36

Passing parameters to Event handlers

- Remember we said if function name is followed with brackets, it gets executed immediately e.g. myFunc().
- A function can return a number, a string , a Boolean **or even a function**!!!
- Yes a function in JS can return a function!!!!
- Based in these two here is an example where parameters were passed to same function handling events for two buttons:

```
<input type="button" value="Hi" id="button1">
<input type="button" value="Hi" id="button2">
<script>
    function handler(a) {
        return function () {
            console.log('Hi ' + a);
        }
    }
    document.getElementById("button1").onclick = handler(1);
    document.getElementById("button2").onclick = handler(2);
</script>
```

This is based on a feature of JavaScript called ***closure***.

Variable a is assessable in the function returned by the enclosing function

向事件处理程序传递参数

- 记得我们说过，如果函数名后面跟有括号，则会立即执行，例如 myFunc()。

- 一个函数可以返回一个数字、一个字符串、一个布尔值 **甚至是一个函数**！！！
- 是的，JavaScript 中的函数可以返回一个函数！！！！
- 基于以上两点，这里有一个示例 p其中参数被传递给同一个函数处理两个按钮事件的函数：

```
<input type="button" value="Hi" id="button1">
<input type="button" value="Hi" id="button2">
<script>
    function handler(a) {
        return function () {
            console.log('Hi ' + a);
        }
    }
    document.getElementById("button1").onclick = handler(1);
    document.getElementById("button2").onclick = handler(2);
</script>
```

这是基于 JavaScript 的一个特性，称为闭包。

变量 a 在外部函数返回的函数内是可访问的

Object constructor with methods (another example was given in lecture 1)

带有方法的对象构造函数（在第1讲中已给出另一个示例）

38

Example: Object constructor with methods (this is old way, you can now use classes in JS)

Q: What does this code snippet do?

```
let arrayButtons = [];
function Button(color, width, height, top, left, order) {
    this.order = order;
    this.btn = document.createElement("button");
    this.btn.style.backgroundColor = color;
    this.btn.style.width = width;
    this.btn.style.height = height;
    this.btn.style.position = "absolute";
    document.body.appendChild(this.btn);
    // A method to set location
    this.setLocation = function (top, left) {
        this.btn.style.top = top;
        this.btn.style.left = left;
    };
    /* we call this method to set original
    top,left during creation of the object*/
    this.setLocation(top, left);

}
arrayButtons.push(new Button("Red", "100px", "100px", "0px", "0px", 0));
arrayButtons.push(new Button("Blue", "200px", "100px", "200px", "200px", 1));
// this has to be in a separate function, e.g. moveRandom ... or something
setInterval(function () {
    arrayButtons[0].setLocation(
        Math.floor(Math.random() * 100) + "px",
        Math.floor(Math.random() * 100) + "px");
}, 500);
```


示例：带有方法的对象构造函数

（这是旧的方式，现在你可以在 JavaScript 中使用类）

问题：这段代码的作用是什么？


```
let arrayButtons = [];
function Button(color, width, height, top, left, order) {
    this.order = order;
    this.btn = document.createElement("button");
    this.btn.style.backgroundColor = color;
    this.btn.style.width = width;
    this.btn.style.height = height;
    this.btn.style.position = "absolute";
    document.body.appendChild(this.btn);
    // A method to set location
    this.setLocation = function (top, left) {
        this.btn.style.top = top;
        this.btn.style.left = left;
    };
    /* we call this method to set original
    top,left during creation of the object*/
    this.setLocation(top, left);

}
arrayButtons.push(new Button("Red", "100px", "100px", "0px", "0px", 0));
arrayButtons.push(new Button("Blue", "200px", "100px", "200px", "200px", 1));
// this has to be in a separate function, e.g. moveRandom ... or something
setInterval(function () {
    arrayButtons[0].setLocation(
        Math.floor(Math.random() * 100) + "px",
        Math.floor(Math.random() * 100) + "px");
}, 500);
```

References

- https://gearheart.io/articles/how-build-scalable-web-applications/
- https://www.cleveroad.com/blog/web-application-architecture
- https://stackify.com/web-application-architecture/
- https://en.wikipedia.org/wiki/Software_architecture
- https://www.jinfonet.com/resources/bi-defined/3-tier-architecture-complete-overview/
- https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html
- https://www.sciencedirect.com/topics/computer-science/presentation-logic
- https://lanars.com/blog/web-application-architecture-101
- ChatGPT was used for proofreading and content generation too

参考文献

- https://gearheart.io/articles/how-build-scalable-web-applications/
- https://www.cleveroad.com/blog/web-application-architecture
- https://stackify.com/web-application-architecture/
- https://en.wikipedia.org/wiki/软件_架构
- https://www.jinfonet.com/resources/bi-defined/3-tier-architecture-complete-overview/
- https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html
- https://www.sciencedirect.com/topics/computer-science/presentation-logic
- https://lanars.com/blog/web-application-architecture-101
- ChatGPT 也被用于校对和内容生成