1. User story

希望資訊能夠清楚明瞭、希望有歷史記錄，觀察趨勢、希望能夠有統整性的資料

希望能及時更新、希望有警示訊息

3.4.5.

檢驗User story

6. UI Design introduction

簡單明瞭的頁面，首頁非常酷，看得舒服

7.

使用上非常直觀、直覺，符合使用者經驗

小結：符合User story 、符合UX

8. Architecture

* 使用React設計和開發前端界面
* 連接後端 API，向後端發送ＰＯＳＴ請求，期望得到一個ＪＳＯＮ檔
* 若後端在cache或資料庫已有資料，直接返回所需資料。否則後端執行爬蟲，爬取所需資料，並存入資料庫，回傳資料給前端。

9. Architecture

10. Demo

11. Code with unit test

* + 測ＡＰＩ有沒有正確的回傳理想的資料
  + 它進行了多個 POST 請求，並驗證了回應的狀態碼和回傳的 JSON 資料是否符合預期。

12. 12 factors - Codebase

* 使用版本控制系統來控管代碼, ex. Github, docker
* 一個應用只對應一個repository(分為前端跟後端)

13.

* 明確聲明各種依賴

14.

* 使用 docker-compose scale 指令可以實現應用程式的可水平擴展性。
* 透過增加或減少容器數量，我們可以根據負載需求調整應用程式的規模。
* 可以看到每一個service都創建了不只一個container

15.

* **當我們在container寫進資料，本機也會保留一份**
* **當process突然掛掉時，儲存的資料不會遺失**
* 左邊是本機目錄、右邊是container目錄
* 這行指令的用途是定義了一個 Db 服務並指定了相關的建置、資料卷和連接埠設定。
* 掛載主機上的目錄到容器內的目錄，使得在容器內部進行資料操作時，同時也在主機上保留了資料的副本。

16.

* 應用程式(Application)不會區別本地或是第三方服務，對它來說兩者都是附加的資源，透過某個 url 或配置的服務定(locator/credentials) 來取得數據
* 若要更改ＧＣＰ換到ＡＷＳ，只需要該改ＩＰ跟ＰＯＲＴ

17.

* 使用config.json來嚴格分離代碼(Code)與配置(Config)
* 建立連線時需要的連線設定

18.

**構建階段（Build）：**

使用Docker容器來創建構建環境，確保構建過程的一致性。可以使用Dockerfile定義構建環境的配置。

**發布階段（Release）：**

將構建好的應用程序打包為Docker鏡像，這個鏡像可以包含應用程序的代碼、依賴項和配置信息。

使用Docker鏡像標籤來區分不同版本的應用程序，例如使用版本號作為標籤。

**運行階段（Run）：**

使用Docker容器運行應用程序的實例。

Run instances of the application in Docker containers.

* docker build 用于构建 Docker 镜像。
* docker push 用于将镜像推送到 Docker Registry。
* docker run 用于在 Docker 上运行容器。

19.

* 透過 port 綁定來提供服務。主要避免撞port及讓應用知道去哪訪問服務。

20.

* 快速啟動、優雅停止
* 收到終止信號(SIGTERM)就會優雅的終止，也就是停止監聽服務的 Port 拒絕所有的請求，執行完當前已請求的服務，然後退出
* 下stop的時候 會送一個stop 的signal 讓container安全結束

21.

* **相同的構建工具**：確保在開發和生產環境中使用相同的構建工具，以避免由於構建過程的不一致性而導致的問題。例如，使用相同版本的構建工具和依賴項管理器。
* **相同的配置管理：**將配置信息存儲在可管理的位置，確保開發和生產環境都使用相同的配置。這可以通過使用配置管理工具（如Ansible、Chef或Puppet）來實現。
* **相同的依賴管理：**確保開發和生產環境使用相同的依賴版本。使用包管理工具（如npm、Maven或pip）來管理依賴項，並使用版本控制系統（如Git）來跟踪依賴項的版本。
* **人員互相交流，使用同樣工具、環境進行開發**

22.

* 通過在運行容器時使用指令**docker logs**可以輕易地查看log。