Husmusen

Protokolldefinition

Version 1.0.0

Andra upplagan, augusti 2024

Inledning

Det här är en kravspecifikation och protokolldefinition av *Husmusen*. *Husmusen* är ett protokoll som specificerar ett databassystem riktat till museum (och liknande institutioner) som vill göra sitt inventarium offentligt sökbart utan att behöva uppfinna/skapa sitt egna system helt från grunden. *Husmusen* är helt enkelt ämnad att ge en grund till ett inventariesystem. Om ni gör ett system som följer den här specifikationen går det bra att kalla ditt system *Husmusenföljsamt*.

Det här protokollet är skapat och skrivet på uppdrag av Bergdala glastekniska museum – ett arbetslivsmuseum i *Glasriket*. Mer information om dem hittas på deras hemsida: https://bergdala-glastekniska-museum.se/

Innehåll

Krav

Det här är de krav som krävs för att implementera *Husmusen*:

- Ett programmeringsspråk som kan:
  - Kommunicera med databasmjukvara.
  - En kan skapa en webbserver med.
- En databasmjukvara. (T.ex. MariaDB, PostgreSQL, MongoDB, etc.)
- En person eller ett team som kan implementera systemet.

Observera!

Husmusen kan implementeras med vilken databasmjukvara som helst bakom kulisserna. Dock kan det vara så att ditt museum kräver användningen av en specifik databasmjukvara. T.ex. kanske museet använder ett webbhotell som bara kan leverera en typ av databasmjukvara.

Funktioner

Det här är alla funktioner som behöver finnas för att ditt system ska vara *Husmusenföljsamt*:

- För offentligheten:
  - Söka efter föremål.
  - Hämta enskilda föremål (via permalänkar).
  - Se alla nyckelord.
- För administratörer och arbetare på museet:
  - Ladda upp nya föremål.
  - Redigera gamla föremål.
  - Markera föremål som utgångna (försvunna/trasiga/o.d.).[1]
  - Ta bort föremål ur databasen.
  - Ladda upp filer.
  - Redigera och ta bort filer.
  - Skapa nya användare.
  - Ändra sitt lösenord.
  - Se server-loggen.
- **Att det följer nedanstående protokollspecifikation.**

✅ Kom ihåg!

För att ett system ska räknas som *Husmusenföljsamt* krävs inte bara att dessa funktioner finns, utan också att de implementeras enligt specifikationen i det här dokumentet.

1 Föremål borde i första hand inte tas bort helt från databasen eftersom metadatan fortfarande kan vara nyttig! Dock, om ett objekt går sönder eller försvinner kan det vara bra att kunna markera det som "utgånget" för att berätta att det inte finns kvar fysiskt.

Datatyper

Det finns en del omfattande datatyper i *Husmusen*: *DBInfo*, *Item*, Keyword, och *File*. Det finns också vissa undertyper, så som *MuseumDetails*, *ItemType* och *ItemData*. Nedan finns en beskrivning av samtliga datatyper.

DBInfo

DBInfo håller information om databassystemet och museet.

```
protocolVersion:         Version
protocolVersions:        String // Komma-separerad bestående av
                                // alla `Version` som är
                                // implementerade
supportedInputFormats:   Array<TransferFormat>
supportedOutputFromats:  Array<TransferFormat>
instanceName:            String
museumDetails:           MuseumDetails
```

Version

En Version formuleras som “siffror punkt siffror punkt siffror”, eller i enlighet med detta RegEx:

```
/\d+\.\d+\.\d+/
```

Nedan följer några xxempel:

```
0.8.2
1.2.2
0.18.4
```

Viktigt!

Versionen **måste** matcha den version av *Husmusen* som är implementerad! Om version 1.0.0 är implementerad så måste det vara den som står i *protocolVersion*-fältet. I *protocolVersions* ska alla implementerade versioner av protokollet listas. Ett system kommer kunna ha stöd för både version 1.0.0 och 1.3.0 via olika URL:er/*routes*:

```
/api/<VERSION>/…
```

Om flera versioner är implementerade ska den **senaste** stå i *protocolVersion*.

Transferformat

TransferFormat är ett format som Husmusen kan ta emot eller skicka information. Följande värden är används i *Husmusen*:

```
"JSON"
"YAML"
```

D.v.s. en *strängrepresentation* av orden *JSON* och *YAML*, som står för *Javascript object notation* respektive *Yet another markup language*.

⚠ Kom ihåg!

Det finns flera olika sätt att skicka data mellan datorer/servrar. De som krävs för att vara *Husmusenföljsamt* är *YAML* och *JSON*. Det skulle gå att implementera fler, t.ex. *XML*, men det är inte nödvändigt för att vara *Husmusenföljsamt*.

❗ Varning!

Det finns ingen standard som beskriver hur andra *TransferFormat* skulle implementeras! Det betyder att om ert museum vill ha ett till format, så måste ni själva skriva en standard för det formatet.

MuseumDetails

MuseumDetails beskriver detaljer om museet, såsom namn och adress.

```
name:        String // Museets namn
description: String // En beskrivning av museet, t.ex.
                    // område, ålder.
address:     String // En address till museet.
location:    String // En mer generell plats, t.ex. ort.
coordinates: String // Koordinater till museet.
website:     String // En URL till museets hemsida.
```

Exempelimplementering: Javascript

```
const MuseumDetails = {
  name:        "Museum",
  description: "Ett museum om museum.",
  address:     "Gatanvägen 4",
  location:    "Kungshamn",
  coordinates: "0°0'0" N, 25°0'0" W",
  website:     "https://example.com"
}
```

⚠ Kom ihåg!

Det här är minimikravet för att vara *Husmusenföljsamt*. Det skulle gå att lägga till fler fält i *MuseumDetails*, men inget av de ovanstående fälten får tas bort eller få sin definition ändrad.

Item

Item är ett föremål, eller koncept/person, som finns i museets databas.

```
name:          String       // Namnet på ett föremål, t.ex. en
                            // boks titel.
description:   String       // En längre beskrivning av
                            // föremålet.
keywords:      String       // Komma-separerad lista med
                            // nyckelord.
type:          ItemType     // Typen av föremål. (Se nedan.)
itemID:        ItemID       // Föremålets unika ID i databasen.
addedAt:       Date         // När objektet blev tillagt i
                            // databasen.
updatedAt:     Date         // När objektet senast blev
                            // uppdaterat.
itemData:      ItemData     // Ett objekts data.
customData:    *            // Här går det att lägga övrig data
                            // som inte är med i standarden.
isExpired:     Boolean      // Beskriver huruvida ett objekt är
                            // markerat som borttaget.
expireReason:  String       // Beskriver anledning till varför
                            // objektet blev markerat som
                            // borttaget.
files:         Array<File>  // En array med filer relaterade
                            // till objektet.
```

ℹ Tips!

I en relationsdatabas kan *ItemData* och *CustomData* lagras som en *JSON-sträng* i ett fält/en kolumn, medan det i en *no-SQL*-databas skulle kunna lagras direkt i samma *objekt/dokument* som ett *objekt/dokument*.

ItemType

Här följer alla giltiga *ItemTypes*.

```
"ArtPiece"
"Blueprint"
"Book"
"Building"
"Collection"
"Concept"
"CulturalEnvironment"
"CulturalHeritage"
"Document"
"Exhibition"
"Film"
"Group"
"HistoricalEvent"
"InteractiveResource"
"Map"
"Organisation"
"Person"
"Photo"
"PhysicalItem"
"Sketch"
"Sound"
```

Dessa speglar de typer som finns med i *K-Samsök*. Typerna återfinns i Riksantikvarieämbetets dokumentation:

https://www.raa.se/hitta-information/k-samsok/att-anvanda-k-samsok/objektstyper/

Viktigt!

I teorin skulle det gå att lägga till fler *ItemTypes* i databasen, men det skulle bryta *Husmusenföljsamheten*! Det är bäst att i första hand använda de *ItemTypes* som är definierade av protokollet, och använda nyckelord för att vidare beskriva föremålet. T.ex. kan en vas beskrivas som ett *PhysicalItem* och sedan vidare beskrivas med nyckelordet *Vas*.

ItemData

ItemData skiljer lite för de olika typerna av objekt. På kommande sidor kommer *ItemData* beskrivas för varje *ItemType*.

Viktigt!

Om det behövs fler fält än de specificerade i *ItemData* ska de fälten läggas i *CustomData* och **inte** läggas till i *ItemData* för att vara *Husmusenföljsamt*!

ArtPiece

ArtPiece är ett konstverk, t.ex. en tavla/målning.

```
artist:   String   // Konstnärens namn
material: String   // Material i konstverket.
style:    String   // Information om konststilen.
weight:   Integer  // Konstverkets vikt i hela gram.
year:     Integer  // Året som konstverket skapades.
```

Blueprint

Blueprint är en ritning.

```
date:     Date    // Datumet då ritningen blev klar.
designer: String  // Personen/-erna som designade/skapade
                  // ritningen.
height:   Integer // Höjden på ritningen i hela millimeter.
scale:    String  // Skalan på ritningen.
width:    Integer // Bredden på ritningen i hela millimeter.
```

Book

Book är en bok.

```
authour:          String  // Namn på bokens författare
ISBN:             String  // Bokens ISBN.
language:         String  // Det språk som boken är skriven på.
originalLanguage: String  // Om boken är översatt, ange
                          // originalspråk.
originalTitle:    String  // Om boken är översatt, ange
                          // originaltitel.
pageCount:        Integer // Antalet sidor i boken.
publisher:        String  // Förlaget som publicerade boken.
title:            String  // Bokens titel.
translator:       String  // Om boken är översatt, ange
                          // översättare.
year:             Integer // Bokens publikationsår.
```

Building

Building är en byggnad.

```
architect:    String  // Byggnadens arkitekt(er).
coordinates: String  // Koordinater till byggnaden.
location:     String  // Generell plats, t.ex. ort.
material:     String  // Materialet på byggnaden, tegel, trä,
                      // cement, etc.
name:         String  // Byggnadens namn.
purpose:      String  // Syftet/typen av byggnad, t.ex. sågverk.
year:         Integer // Året som byggnaden blev klar.
```

Collection

Collection är en samling av saker.

```
collectible: String  // Saken som samlas.
collector:   String  // Personen/-arna som samlar.
size:        Integer // Antalet saker i samlingen.
```

Concept

Inte definierat. Använd *customData* för att beskriva föremålet.

CulturalEnvironment

CulturalEnvironment är en kulturell miljö.

```
coordinates: String // Koordinater till miljön.
location:    String // Generell plats, t.ex. ort.
name:        String // Namnet på miljön.
```

CulturalHeritage

CulturalHeritage är en kulturlämning, som t.ex. skeppssättning, runsten, eller en kyrka.

```
coordinates: String // Koordinater till lämningen.
location:    String // Generell plats, t.ex. ort.
name:        String // Namnet på lämningen.
type:        String // Typ av lämning, t.ex. runsten.
```

Document

Document är ett dokument, t.ex. rapport, pamflett, manifest.

```
authour:          String  // Dokumentets författare.
documentType:     String  // Typ av dokument, t.ex. manifest,
                          // protokoll, brev.
language:         String  // Dokumentets språk.
originalLanguage: String  // Om dokumentet är översatt,
                          // originalspråket.
originalTitle:    String  // Om dokumentet är översatt,
                          // originalspråket.
publisher:        String  // Person/organisation som
                          // publicerade dokumentet.
title:            String  // Dokumentets titel.
translator:       String  // Om dokumentet är översatt,
                          // översättaren.
year:             Integer // Publikationsåret.
```

Exhibition

Exhibition är en utsällning.

```
coordinates: String // Koordinaterna till utställningen.
exhibit:     String // Det som ställs ut.
location:    String // Generell plats, t.ex. museum/park/ort.
name:        String // Namn på utställningen.
organiser:   String // Den person/organisation som organiserar
                    // utställningen.
```

Film

Film är en film eller video.

```
director: String  // Regisören.
language: String  // Språket/språken i filmen.
subject:  String  // Ämnet som filmen hanterar.
title:    String  // Filmens titel.
type:     String  // Typ av film, ex. spelfilm, dokumentär.
writer:   String  // Manusförfattaren.
year:     Integer // Året som filmen släpptes.
```

Group

Group är en grupp människor.

```
goal:        String  // Målet/syftet med gruppen, t.ex. Parti,
                     // musikband.
memberCount: Integer // Antalet medlemmar.
members:     String  // Framstående medlemmar.
name:        String  // Gruppens namn.
```

HistoricalEvent

HistoricalEvent syftar till en historisk händelse.

```
date: Date   // Datumet då händelsen skedde.
name: String // Namnet på händelsen.
type: String // Typ av händelse, t.ex. krig, revolution.
```

InteractiveResource

En *InteractiveResource* är någon man kan interagera med, t.ex. simulation på en hemsida.

```
uri:         String // Länk till resursen. (Om digital.)
location:    String // Resursens plats.
coordinates: String // Koordinater till platsen.
```

Map

Map är en karta.

```
area:          String  // Den yta som visas av kartan.
chartographer: String  // Den som skapade/ritade kartan.
year:          Integer // Året som kartan ritades.
scale:         String  // Kartans skala.
width:         Integer // Kartans bredd i millimeter.
height:        Integer // Kartans höjd i millimeter.
```

Organisation

En *Organisation* är en organisation.

```
goal:        String  // Målet/syftet med organisation, t.ex.
                     // välgörenhet.
memberCount: Integer // Antalet medlemmar.
members:     String  // Framstående medlemmar.
name:        String  // Organisationens namn.
```

Person

En *Person* är en person.

```
firstName:   String // Personens förnamn.
middleNames: String // Personens mellannamn.
lastName:    String // Personens efternamn.
alias:       String // Personens alias/smeknamn/nome de plume.
occupation:  String // Personens yrke/uppdrag.
```

Photo

Photo är ett fotografi.

```
photographer: String // Namn på fotografen som tog bilden.
subject:      String // Motivet på bilden.
type:         String // Typ av foto, t.ex. porträtt, naturbild.
date:         Date   // Datumet som fotot togs.
```

PhysicalItem

PhysicalItem är en fysisk sak, t.ex. en vas.

```
creator:  String   // Skaparen av föremålet.
type:     String   // Typ av föremål, t.ex. vas.
material: String   // Material i föremålet.
style:    String   // Information om konststilen.
weight:   Integer  // Konstverkets vikt i hela gram.
year:     Integer  // Året som föremålet skapades.
```

Sketch

Sketch är en skiss/teckning.

```
artist:   String   // Konstnärens namn.
style:    String   // Information om konststilen.
subject:  String   // Vad skissen/teckningen föreställer.
year:     Integer  // Året som konstverket skapades.
```

Sound

Sound är ett ljud, en inspelning (t.ex. radioprogram), eller musik.

```
type:        String  // Om det är musik, radioprogram, etc.
voices:      String  // Vilka röster som finns med.
instruments: String  // Vilka instrument som finns med.
duration:    Integer // Ljudets längd i sekunder.
```

ItemID

ItemID ska vara ett unikt ID som varje föremål har. **Det ska aldrig ändras någon gång!** Detta ID ska användas för att skapa *permalänkar* till föremålet.

✅ Kom ihåg!

Syftet med *ItemID* är att ge varje objekt en unik permalänk. Det finns olika sätt att skapa unika ID:n, t.ex. unika nummer (heltal), *UUIDs*, eller *GUIDs*.

ℹ️ Tips!

Ett sätt att implementera unika ID:n skulle vara att det första objektet får ID:t 1, det andra 2, det tredje 3, osv.

Keyword

Ett *Keyword* beskriver ett nyckelord.

```
type:        ItemType // Typen av föremål nyckelordet är för,
                      // t.ex. `Document`
word:        String   // Nyckelordet självt, t.ex. "Vas"
description: String   // En beskrvning av när nyckelordet ska
                      // användas.
```

File

En *File* beskriver en fils metadata.

```
name:        String   // Namnet på ett filen, t.ex. titel
description: String   // En beskrivning i filen.
type:        FileType // Typen av fil. (Se nedan.)
license:     String   // Den licens som filen har.
fileID:      FileID   // Filens unika ID i databasen.
addedAt:     Date     // När filen blev tillagd i databasen.
updatedAt:   Date     // När filen senast blev redigerad i
                      // databasen.
relatedItem: ItemID   // `ItemID` av det objekt som filen är
                      // tillhör.
```

Viktigt!

Det är **inte bra** att lagra själva fildatan i en databasmjukvara. Själva datan bör istället lagras i molnet eller serverns filsystem.

FileID

FileID ska vara ett unikt ID som delas ut till varje fil i databasen, likt *ItemID* så kan det vara utformat på flera sätt, t.ex. unika heltal, *UUIDs*, eller *GUIDs*.

Databasspecifikation (MariaDB)

På nästa sida följer ett exempel på hur en databasstruktur kan skapas i databasmjukvaran MariaDB för att hålla *Items*. Exemplet använder en *ENUM* som används för att förutbestämma en lista med olika giltiga värden.

```
CREATE TABLE husmusen_items (
    itemID        INTEGER      PRIMARY KEY,
    name          VARCHAR(128) NOT NULL,
    keywords      TEXT         DEFAULT '',
    description   TEXT         DEFAULT '',
    type          ENUM(
        "ArtPiece",
        "Blueprint",
        "Book",
        "Building",
        "Collection",
        "Concept",
        "CulturalEnvironment",
        "CulturalHeritage",
        "Document",
        "Exhibition",
        "Film",
        "Group",
        "HistoricalEvent",
        "InteractiveResource",
        "PhysicalItem",
        "Map",
        "Organisation",
        "Person",
        "Photo",
        "Sketch",
        "Sound"
    ),
    addedAt      TIMESTAMP    DEFAULT current_timestamp,
    updatedAt    TIMESTAMP    DEFAULT current_timestamp,
    itemData     JSON         DEFAULT '{}' CHECK
                (JSON_VALID(itemData)),
    customData   JSON         DEFAULT '{}' CHECK
                (JSON_VALID(customData)),
    isExpired    BOOLEAN      DEFAULT 0,
    expireReason TEXT         DEFAULT ''
);
```

Tips!

För att öka prestandan i databasen går det att använda *indexes*. *Indexes* är datastrukturer som snabbar på läsning och upphittande av diverse data.

Tips!

Det finns databaser som liknar *MariaDB*, t.ex. *PostgreSQL* och *MySQL*. De borde ha liknande sätt att skapa tabellen, men det finns små detaljer som skiljer dem åt.

API-definition

Alla *endpoints* som är markerade med ett lås (🔒) ska vara skyddade med ett säkerhetssystem så att endast personer som ska kunna modifiera databasen kan nå dem. *Endpoints* markerade med en flagga (⚐) ska bara kunna nås av databasens administratörer. Detta säkerhetssystem beskrivs senare.

Det går att använda *Husmusen-Output-Format-headern* för att specificera vilken typ av *TransferFormat* som önskas.

OBS! I *Husmusen-Output-Format-headern* står formaten enligt deras *mime types*, t.ex. används application/json för att specificera JSON. Om inget är specificerat så impliceras det vara JSON.

För att specificera vilken *TransferFormat* som används när du skickar information till servern används *Content-Type-headern*.

OBS! Även i *Content-Type-headern* står formaten enligt deras *mime types*, t.ex. används application/json för att specificera JSON. Om inget är specificerat så impliceras det vara JSON.

DBInfo

Här följer alla *endpoints* som har att göra med *DBInfo*.

GET: /api/db_info

Skickar tillbaka *DBInfo*. Exempel:

```
{
  "protocolVersion": "1.0.0",
  "protocolVersions": [ "1.0.0", "0.8.0" ],
  "supportedInputFormats": [ "YAML", "JSON" ],
  "supportedOutputFormats": [ "YAML", "JSON" ],
  "instanceName": "Husmusen på Museum",
  "museumDetails": {
    "name": "Museum",
    "description": "Ett helt vanligt museum.",
    "address": "Gatanvägen 4",
    "location": "Kungshamn",
    "coordinates": "0°0'0" N, 25°0'0" W",
    "website": "https://example.com"
  }
}
```

GET: /api/db_info/version

Skickar tillbaka den **senaste** versionen som implementeringen har stöd för. Exempel:

```
1.0.0
```

⚠ Observera!

Den här information ska **alltid** skickas tillbaka som *text/plain*!

GET: /api/db_info/versions

Skickar tillbaka de versionerna som implementeringen har stöd för. (Ska vara sorterad med senaste versionen först.) Exempel:

```
1.0.0,0.8.0
```

⚠ Observera!

Den här information ska **alltid** skickas tillbaka som *text/plain*! Varje version ska vara separerad med ett kommatecken.

POST: /api/db_info 🔒 ⚐

Används för att ändra databasinformation. I förfrågan ska **hela** (inte bara det som ändrats) den nya databasinformationen stå med. Exempel:

```
{
  "instanceName": "Husmusen på Museum",
  "museumDetails": {
    "name": "Museum",
    "description": "Ett helt vanligt museum.",
    "address": "Gatanvägen 4",
    "location": "Kungshamn",
    "coordinates": "0°0'0" N, 25°0'0" W",
    "website": "https://example.com"
  }
}
```

❗ Varning!

ProtocolVersion, *ProtocolVersions*, *SupportedInputFormats*, och *SupportedOutputFormats* ska inte gå att ändra på detta vis! De ska specificeras av servern själv. Programmeraren/-arna kan t.ex. specificera dem i en fil i serverns kod och den filen läses för att veta vilka versioner och *TransferFormat* som stöds.

Item

Här följer alla *endpoints* som har med *Items* att göra.

GET: /api/1.0.0/item/search

Används för att söka efter föremål i databasen. Svaret består av en *array* med alla *Items* som matchar sökningen. Om inget föremål matchar sökningen så är *arrayen* tom. Denna *enpoint* tar en del *queries*/parametrar[2]. Nedan följer en beskrivning av alla parametrar.

?types

?types-parametern används för att beskriva vilka *ItemTypes* som användaren vill söka efter. Den består av en kommaseparerad lista av just *ItemTypes*. Om det ska sökas efter böcker och dokument skulle det se ut såhär:

```
/api/1.0.0/item/search?types=Book,Document
```

?freetext

?freetext-parametern används för att söka i namn och beskrivning av föremål. Om en användare vill söka efter *glasvas* skulle det se ut såhär:

```
/api/1.0.0/item/search?freetext=glasvas
```

Exakt implementering av fritextsökningen är dessvärre inte en del av detta protokoll ännu. Det eftersom fritextsökning kan skilja lite i olika databasmjukvaror och att det inte finns en standard skapad för *Husmusen* ännu.

?keywords

?keywords-parametern används för att specificera vilka nyckelord som användaren önskar söka efter. Precis som *?types*-parametern så specificeras nyckelorden genom en kommaseparerad lista.

Tips!

Det går att rensa ogiltiga nyckelord bakom kulisserna, så slipper databasen söka efter nyckelord som inte finns.

2 Alltså de värden som följer ett frågetecken (?) i en URL. Det ser ut såhär i en URL: https://example.com?query1=value1&query2=anothervalue.

?keyword-mode

?keyword_mode-parametern väljer vilket läge som nyckelordssökningnen ska använda. Det finns två lägen: *OR* och *AND*. *OR*-läget fungerar på så sätt att resultaten måste innehålla **minst ett** av de specificerade nyckelorden. *AND*-läget fungerar på så sätt att alla resultat måste innehålla **samtliga** nyckelord. Om denna parameter inte specificeras faller den tillbaka på *OR*.

?sort

?sort-parametern används för att välja vilket fält som resultaten ska sorteras efter. Giltiga värden är: *relevance*, *name*, *addedAt*, *updatedAt*, och *itemID*. Om denna parameter inte specificeras så faller den tillbaka på *name*.

⚠ Observera!

Relevans (*relevance*) sorterar sökresultaten från hög relevans till låg relevans, alltså sjunkande. Alla andra sorteringsfält är i stigande ordning.

?reverse

?reverse-parametern används för att välja om resultatens ordning ska vändas i bakvänd ordning. Giltiga värden är *1*, *0*, *on*, *off*, *true*, och *false*. *1*, *on*, och *true* används för att vända ordningen. Om denna parameter inte specificeras så faller den tillbaka på *off*.

Exempel: böcker om jordbruk

Om en användare vill söka efter böcker om jordbruk och sortera dem i alfabetisk ordning, skulle förfrågan behöva se ut såhär:

```
/api/1.0.0/item/search?freetext=jordbruk&types=Book
&sort=alphabetical
```

För att URL:en ska få plats har den en radbrytning, men såklart skulle den inte ha det i verkliga livet, eftersom en URL inte kan ha radbrytningar.

GET: /api/1.0.0/item/info/<id>

Används för att hämta information om ett *Item* utefter dess *ItemID*. Låt säga att en användare önskar hämta föremål *5612*, då skulle URL:en se ut såhär:

`/api/1.0.0/item/info/5612`

Resultatet är ett *Item* och skulle kunna se ut såhär:

```
{
  "name": "Utvandrarna",
  "description": "Bok av Vilhelm Moberg.",
  "keywords": "Småland,Amerika,1800-talet",
  "type": "Book",
  "itemID": 5612,
  "addedAt": "1970-01-01T00:00:00.000Z",
  "updatedAt": "2001-09-09T01:46:40.000Z",
  "itemData": {
    "...": "..."
  },
  "customData": null,
  "hasThumbnail": true,
  "itemFiles": [
    "File1",
    "File2",
    "...",
  ]
}
```

POST: /api/1.0.0/item/new 🔒

Används för att stoppa in ett nytt *Item* i databasen. Förfrågan består av metadata som beskriver föremålet. Exempel:

```
{
  "name": "Utvandrarna",
  "description": "Bok av Vilhelm Moberg.",
  "keywords": "Småland,Amerika,1800-talet",
  "type": "Book",
  "itemData": {
    "...": "..."
  },
  "customData": null,
}
```

⚠ Observera!

Lägg märke till att det inte går att specifiera ett föremåls *itemID*, *addedAt*, *updatedAt*, eller *files*. De skall antingen fixas automatiskt av servern när föremålets skapas eller genom att skapa en fil och länka ihop det med föremålet.

POST: /api/1.0.0/item/edit 🔒

Används för att ändra/uppdatera på det föremål i databasen vars ID är specifierat i förfrågan. Du skickar helt enkelt en förfrågan med den nya metadatan. Exempel:

```
{
  "itemID": 432
  "name": "Utvandrarna",
  "description": "Bok av Vilhelm Moberg.",
  "keywords": "Småland,Amerika,1800-talet",
  "type": "Book",
  "itemData": {
    "...": "..."
  },
  "customData": null,
}
```

⚠ Observera!

Lägg märke till att det inte går att specifiera ett föremåls *itemID*, *addedAt*, *updatedAt*, eller *files*. De skall antingen fixas automatiskt av servern när föremålets skapas eller genom att skapa en fil och länka ihop det med föremålet.

POST: /api/1.0.0/item/mark 🔒

Används för att markera föremålet *id* som utgånget, t.ex. om ett föremål har försvunnit eller gått sönder. I förfrågan behöver en anledning finnas med som förklarar varför objektet blir markerat som "borttaget", ex:

```
{
  "itemID": 342
  "reason": "Den här vasen har gått sönder."
}
```

⚠ Kom ihåg!

Objektet ska inte faktiskt plockas bort från databasen eftersom metadatan fortfarande kan vara viktig/intressant för allmänheten, även om det fysiska föremålet har gått sönder.

POST: /api/1.0.0/item/delete 🔒 ⚐

Används för att permanent ta bort det föremål ur databasen vars ID specificeras i förfrågan.

```
{
  "itemID": 342
}
```

Observera!

I första hand bör saker markeras som utgångna, men om ett föremål faktiskt behöver tas bort av någon anledning, så behöver det ändå kunna göras.

Kom ihåg!

De filer som är relaterade till föremålet bör också tas bort! Annars kan de hittas med hjälp av permalänkar!

Files

Här följer alla *endpoints* som har med *Files* att göra.

GET: /api/1.0.0/file/get/<id>

Används för att hämta själva filen. Om en fil med ID:t *345* ska hämtas så använder du helt enkelt:

```
/api/1.0.0/file/get/345
```

GET: /api/1.0.0/file/file/<id>

Används för att hämta filens metadata. Om en fil med ID:t *345* ska hämtas så använder du helt enkelt:

```
/api/1.0.0/file/info/345
```

Resultatet skulle ser ut såhär:

```
{
  "name": "Fågelsång - Blåmes",
  "type": "audio/mp3",
  "license": "CC BY-SA 3.0",
  "itemID": 345,
  "addedAt": "...",
  "relatedItem": 4791
}
```

Den här filen är länkad till föremålet med ID:t *4791*. Det går att se genom *RelatedItem*-fältet som har värdet *4791*.

POST: /api/1.0.0/file/new 🔒

Används för att skapa en ny fil. Om en ny fil ska skapas ska användaren skicka en *POST-request* till:

`/api/1.0.0/file/new`

I förfrågan behöver all metadata beskrivas:

```
{
  "name": "Fågelsång - Blåmes",
  "license": "CC BY-SA 3.0",
  "relatedItem": 4791,
  "dataBuffer": "..."
}
```

⚠ Observera!

Lägg märke till att det inte går att specificera en fils *fileID*, *type*, eller *addedAt*. De ska fixas automatiskt av servern när filen läggs till i databasen.

POST: /api/1.0.0/file/edit 🔒

Används för att ändra/uppdatera metadatan på den fil vars ID är specifierat i förfrågan. I förfrågan behöver all ny metadata beskrivas:

```
{
  "fileID": "64291c17-c16d-4cff-b590-2043bf89981d"
  "name": "Fågelsång - Blåmes",
  "license": "CC BY-SA 3.0",
  "relatedItem": 4791
}
```

⚠ Observera!

Lägg märke till att det inte går att specificera en fils *fileID*, *type*, eller *addedAt*. De ska fixas automatiskt av servern när filen läggs till eller uppdateras.

POST: /api/1.0.0/file/delete/ 🔒

Används för att ta bort filen vars ID specificeras i förfrågan.

```
{
  "fileID": "64291c17-c16d-4cff-b590-2043bf89981d"
}
```

⚠ Kom ihåg!

Själva filen bör också plockas bort, och inte bara metadatan i databasen.

Keywords

Här följer alla *endpoints* som har med *Keywords* att göra.

GET: /api/1.0.0/keyword

Skickar tillbaka en *array* med alla *Keywords* som finns definierade i databasen. Resultatet ser ut såhär:

```
[
  {
    "type": "Book",
    "word": "Artikel",
    "description": "Vetenskaplig artikel, populärvetenskaplig
artikel eller artikel i dagstidning eller magasin."
  },
  {
    "type": "Book",
    "word": "Avhandling-Doktor",
    "description": "Akademisk avhandling för doktorsexamen."
  },
  {
    "type": "Book",
    "word": "Avhandling-Licentiat",
    "description": "Akademisk avhandling för licentiatsexamen."
  },
  {
    ...
  },
  ...
]
```

GET: /api/1.0.0/keyword/‹types›

Skickar en *array* med alla nyckelord för *types*. *types* är en kommaseparerad lista med alla types som användaren vill ha nyckelord från. Om alla nyckelord för böcker och dokument efterfrågas skulle URL:en se ut såhär:

```
/api/1.0.0/keyword/Book,Document
```

POST: /api/1.0.0/keyword

Används för att låta en administratör konfigurera vilka nyckelord som finns och kan användas i databasen. Förfrågan består av en *array* av alla nyckelord som ska finnas med, även de som fanns innan. Den nya listan med nyckelord skriver alltså över den föregående. Exempel på förfrågan:

```
[
  {
    "type": "Book",
    "word": "Artikel",
    "description": "Vetenskaplig artikel, populärvetenskaplig
artikel eller artikel i dagstidning eller magasin."
  },
  {
    "type": "Book",
    "word": "Avhandling-Doktor",
    "description": "Akademisk avhandling för doktorsexamen."
  },
  {
    "type": "Book",
    "word": "Avhandling-Licentiat",
    "description": "Akademisk avhandling för licentiatsexamen."
  },
  {
    ...
  },
  ...
]
```

Log

Här följer funktionen som låter en administratör se serverloggen. Exakt vilka meddelanden som ska finnas med i loggen är upp till implementeringen, men ett meddelande i loggen ska se ut såhär:

```
prefix:    ItemType // Vilken del av servern som meddelandet
                    // kommer ifrån. T.ex. "auth", "database"
timestamp: String   // Tidpunkten då meddelandet skickades.
                    // I formatet DD/MM/ÅÅÅÅ TT:MM:SS
message:   String   // Meddelandet som skickas i loggen.
```

GET: /api/1.0.0/log/get

Skickar tillbaka en *array* av loggmeddelanden i enlighet med strukturen som beskrevs ovan. Här följer ett exempel på hur resultatet skulle kunna se ut:

```
[
  {
    "prefix": "Log",
    "timestamp": "29/9/2022 21:22:11",
    "message": "Admin 'admin' accessed the log!"
  },
  {
    "prefix": "Auth",
    "timestamp": "29/9/2022 21:23:43",
    "message": "Admin accessed path '/api/1.0.0/log/get'!"
  },
  {
    "prefix": "Log",
    "timestamp": "29/9/2022 21:23:43",
    "message": "Admin 'admin' accessed the log!"
  }
]
```

?reverse

?reverse-parametern kan användas för att vända loggen på andra hållet. Det kan vara användbart om en vill ha de senaste loggmeddelandena först istället för längst bak i resultatet.

Säkerhetssystem

Vissa *enpoints* hanterar antingen potentiellt känslig information (server loggen) eller funktioner som inte vem som helst ska kunna använda, såsom att skapa eller redigera föremål. Dessa *enpoints* behöver därför vara skyddade. Det görs med hjälp av *Husmusen-Access-Token-headern*. Den behöver innehålla någon form av identifierare som beskriver en användare som kan behandla databasen. Det finns två typer av dessa användare: *vanliga* användare och administratörer. Administratörer har vissa privilegier som *vanliga* användare inte har. Denna *header* behöver vara med i alla förfrågningar till skyddade *enpoints* och behöver alltid valideras.

Ett sätt att implementera dessa identifierare är med hjälp av så kallade *JSON Web Tokens*. Dock är det helt upp till implementeringen. Härefter följer information om hur användare skapas eller tas bort, ändrar sitt lösenord, loggar in, och så vidare.

POST: /api/auth/login

Används för att logga in och skaffa en identifierare/*token*. Förfrågan behöver bestå av ett användarnamn och ett lösenord:

```
username: String // Användarnamnet
password: String // Lösenordet
```

Här är ett exempel på hur förfrågan kan se ut:

```
{
  "username": "Användarnamn",
  "password": "SuperbraLösenord"
}
```

Svaret skulle se ut såhär, förutsatt att användarnamn och lösenord är rätt:

```
{
  "token": "xxxx",
  "validUntil": "2022-11-09T22:18:26.625Z"
}
```

Om identifierarna är tidsbegränsade – vilket de borde av säkerhetsskäl – så kan *validUntil*-fältet användas för att beskriva hur länge identifieraren kommer att vara giltig. Detta ska vara beskrivet enligt *ISO 8601*.

Viktigt!

Denna information bör inte skickas över det vanliga *HTTP-protokollet* utan borde skickas över det säkra (krypterade) *HTTPS-protokollet*. Detta eftersom informationen innehåller känslig information, användarnamn och lösenord.

POST: /api/auth/who

Denna *enpoint* kan användas för att ta reda på om en identifierare fortfarande är giltig. Denna *enpoint* skickar tillbaka användarens namn och om den är admin, förutsatt att den har en giltig identifierare.

Svaret ser ut såhär:

```
{
  "username": "admin",
  "isAdmin": true
}
```

Observera!

Beroende på hur *true* lagras i databasen kan detta antigen komma tillbaka som *true* eller som *1* (en etta). Båda är giltiga.

POST: /api/auth/new

Denna *enpoint* används för att skapa en ny användare. Förfrågan behöver se ut såhär:

```
username: String  // Användarens användarnamn.
password: String  // Användarens lösenord.
isAdmin:  Boolean // Beskriver om användaren ska vara admin.
```

Om en användare *Bob* ska skapas med lösenord *Makaron78*, samt ska vara en administratör, skulle förfrågan se ut såhär:

```
{
  "username": "Bob",
  "password": "Makaron",
  "isAdmin": true
}
```

POST: /api/auth/change_password 🔒

Denna *endpoint* låter en användare ändra sitt lösenord. Förfrågan behöver innehålla det gamla och det nya lösenordet:

```
currentPassoword: String  // Det nuvarande lösenordet.
newPassword:      String  // Det nya lösenordet.
```

Om Bob nu vill ändra sitt lösenord från *Makaron78* till *Spaghetti87*, skulle förfrågan se ut såhär:

```
{
  "currentPassword": "Makaron78",
  "newPassword": "Spaghetti87"
}
```

POST: /api/auth/debug_admin_creation

Denna *enpoint* ska bara vara tillgänglig i *debug mode* – ett läge då servern kan ha mer verbosa loggar och extra funktioner. Just denna funktion låter **vem som helst** skapa en initial administratör till databasen. Förfrågan behöver bestå av ett användarnamn och ett lösenord:

```
username: String // Användarnamnet
password: String // Lösenordet
```

Här är ett exempel på hur förfrågan kan se ut:

```
{
  "username": "Användarnamn",
  "password": "SuperbraLösenord"
}
```

Varning!

Denna funktion ska **aldrig** vara påslagen mer än i testscenarion eller för att skapa en initial administratör. Denna funktion kan vara destruktiv om någon som inte ska ha tillgång till den använder den!

Framåt- och bakåtkompatibilitet

För att protokollet ska kunna hållas framåt- och bakåtkompatibelt behöver *API-versionen* läggas till i URL:en på följande vis:

```
/api/<VERSION>/...
```

Detta gör att nya funktioner kan läggas till, men ändå behålla de gamla funktionerna. Detta kan vara viktigt av historiska skäl. Dock finns det vissa funktioner i protokollet som inte ligger bakom en version i URL:en. Det eftersom de förväntas att *alltid* fungera. T.ex. måste en klient/användare alltid kunna nå */api/db_info* för att ta reda på vilka versioner som stöds.

Om versionen benämns *X.Y.Z* så ändras siffrorna utefter:

Z	Om en liten sak ändras som egentligen inte påverkar protokollet så mycket. T.ex. kan det handla om att en ny icke-obligatorisk parameter läggs till till sökfunktionen.
Y	Om något substantiellt läggs till i protokollet, men inget gammalt ändras. T.ex. kanske det läggs till en ny sökfunktion, men den gamla sökfunktionen finns kvar och påverkas inte.
X	Om gamla funktioner omdefinieras så att de ser annorlunda ut. T.ex. kanske den nuvarande sökfunktionen byts ut mot en ny, och datatyperna omdefinieras så att de ser annorlunda ut.

Errorhantering

För det mesta använder protokollet *HTTP-statuskoder* för att berätta om det skett ett error och i så fall vilken typ av error som skett, men det finns också vissa errorkoder specifika till *Husmusen*.

HTTP-statuskoder

Här är några *HTTP-statuskoder*, men inte alla. Dessa är bara exempel på några vanliga som kan vara användbara. Information om andra statuskoder kan hittas här: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

Kod	Beskrivning
200	OK; inga error.
400	Dålig förfrågan. T.ex. kanske en parameter saknas.
404	Resursen hittades inte. Resursen finns inte.
500	Internt serverfel. Något gick fel i servern.

Husmusen-errorkoder

Dessa koder kan användas i ett svar för att mer precist visa vad som gick fel.

Kod	Beskrivning
ERR_UNKNOWN_ERROR	Okänt eller ospecificerat error.
ERR_OBJECT_NOT_FOUND	Ett föremål hittades inte.
ERR_FILE_NOT_FOUND	En fil hittades inte.
ERR_USER_NOT_FOUND	En användare hittades inte.
ERR_MISSING_PARAMETER	En parameter saknas.
ERR_INVALID_PARAMETER	En parameter är ogiltig.
ERR_ALREADY_EXISTS	Det som försöker skapas finns redan.
ERR_DATABASE_ERROR	Något gick fel med databasmjukvaran.
ERR_FILESYSTEM_ERROR	Något gick fel med databasens filsystem.
ERR_INVALID_PASSWORD	Felaktigt lösenord.
ERR_FORBIDDEN_ACTION	Det som försökte göras är otillåtet.

Ett error är strukturerat såhär:

```
errorCode:        String  // Errorkoden. (Finns beskrivna på
                          // föregående sida.)
errorDescription: String  // En mer djupgående förklaring av vad
                          // som har hänt.
```

Om föremålet *345* ska skapas, men redan finns, så kan ett error se ut såhär:

```
{
  "errorCode": "ERR_ALREADY_EXISTS",
  "errorDescription": "The item with ID '345' already exists!"
}
```

Avslutande ord

Antagligen finns det saker som saknas i denna protokolldefinition. Om det finns några frågor eller förslag går det att skicka in dem som *issues* via kod-delningsplattformen *GitHub* här: https://github.com/mushuset/docs

Den här protokolldefinitionen har behandlat version 1.0.0 av protokollet *Husmusen* vars syfte är att skapa ett digitalt inventariesystem för museum och liknande institutioner. Det här är första upplagan.

Exempelimplementeringar

Det finns två exempelimplementeringar av *Husmusen* skrivna i *Javascript* (*Node*) respektive *PHP* (*Laravel*). De finns tillgänglig här:

- https://github.com/mushuset/husmusen
- https://github.com/mushuset/husmusen-php

Uppdrag åt framtiden

Kompatibilitet med K-Samsök

Husmusen är förhoppningsvis redan *kompatibelt nog* med Riksantikvarieämbetets *K-Samsök* – ett system som samlar metadata från museum och liknande institutioner och gör dem sökbara – och om *Husmusen-protokollet* inte är det behöver det göras kompatibelt. I vilket fall bör en guide skrivas som beskriver processen att integrera *Husmusen* med *K-Samsök*.

Bättre/fler exempel och errorkoder

Detta dokument skulle gynnas av fler och bättre exempel. Protokollet i sig skulle också gynnas av fler och bättre errorkoder.

Husmusen 2.0?

Under arbetet med att bygga Husmusen i PHP kändes det som att protokollet inte är så robust och tydligt som det borde vara. Därav borde arbetet med en version två påbörjas någon gång i framtiden.

Revideringshistorik

Första upplagan – oktober 2022

Första upplagan.

Andra upplagan – augusti 2024

✎ Korrigering av viss API-definition som var ”fel”; d.v.s. inte överensstämde med exempelimplementeringen i Node.[3]

✎ Bytt ut vissa långa tankstreck (—) mot korta tankstreck (–) för att följa svenska skrivregler.

✎ Ändrat sidlayouten aningen. Den är nu mer lik en bok.

⊞ Lagt till information om exempelimplementeringar. (Specifik tillägg om en exempelimplementering i PHP.)

⊟ Likaså togs målet att skapa fler exempelimplementeringar (främst i PHP) bort från ”Uppdrag åt framtiden”.

3 Det skulle gå att argumentera för att exempelimplementeringen istället skulle ändras, men då jag inte har kännedom om något annat museum som har implementerat detta protokoll så är detta den enklaste (men kanske ändå inte bästa) lösningen.