Dataccount System Overview

Problem statement

Many companies maintain databases of personal information about individuals, which they use for their own purposes:

- Web sites like Facebook, Amazon, and Google monitor your online activities, then sell the information to advertisers for their own profit, with no compensation to you.
- Roomba's robot vacuums make a map of the interior of your home and send it back to the company. Roomba uses it to improve their robot algorithms, again with no compensation.
- Amazon keeps a permanent database of your personal home address and credit card information, which they can use as they see fit.
- Doctors keep your sensitive medical history in their own databases. You have no control over it.

Thus, the first problem is that information about yourself is stored by many companies, who generally promise to keep it safe, but you have no concrete way of verifying it. The companies may accidentally expose information which is private or embarrassing, or which allows other people to impersonate you. They can make money from your information without compensating you. You have to trust them.

The second problem is on the other side. Companies are exposed scandal, lawsuits, etc. if they lose their customer's information, for example to hackers.

Proposed solution

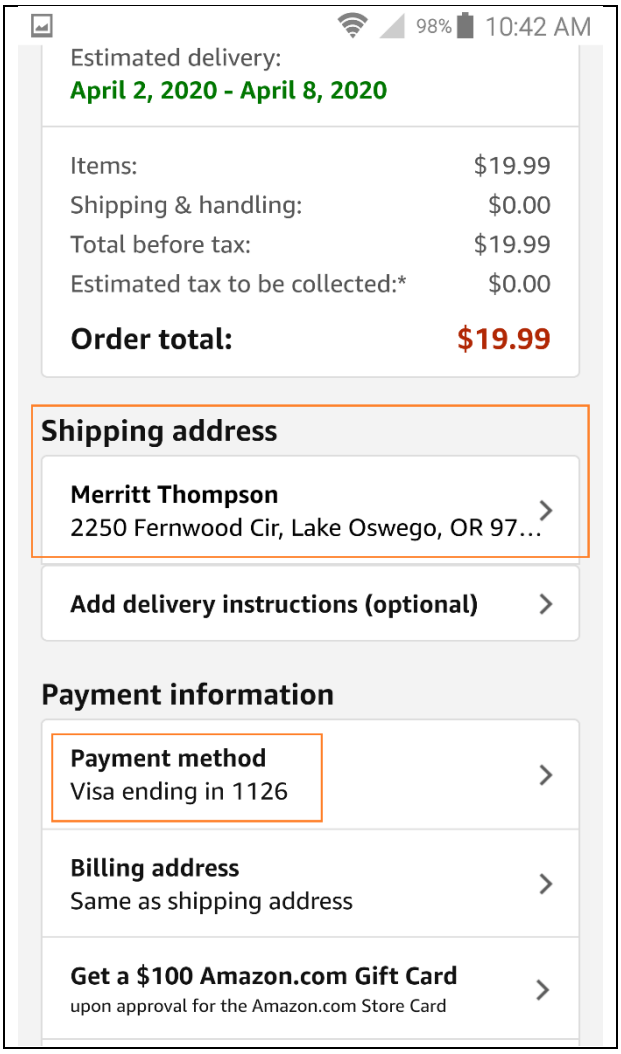
This document proposes a combination of computer systems and laws that would:

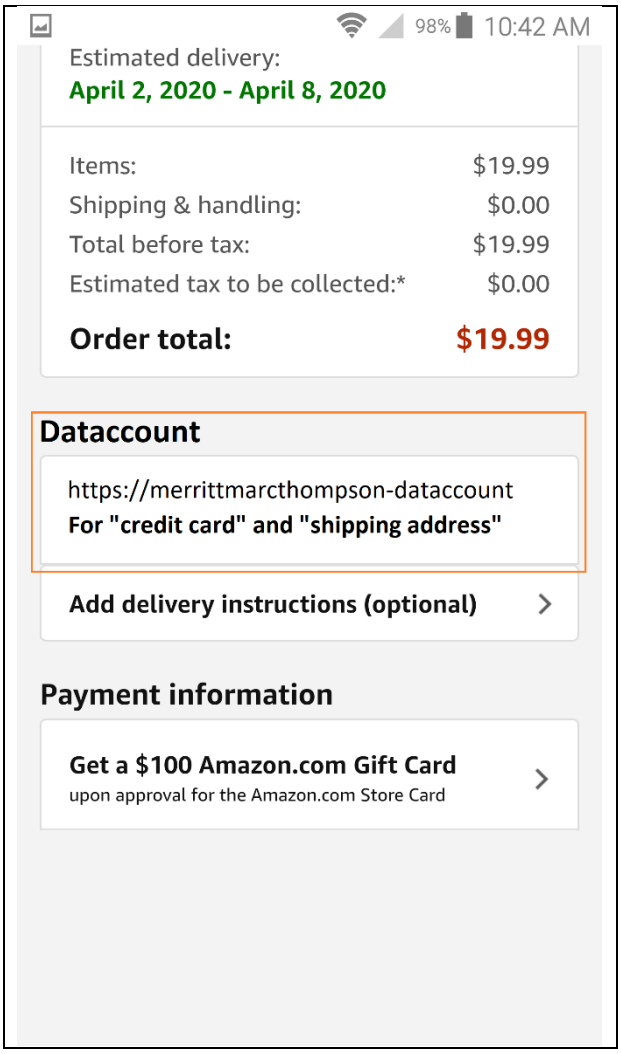
A. Allow companies to access personal information under the control of the individual.
B. Allow individuals to track accesses to their own personal information.
C. Prevent companies from storing personal information.

A. Allowing access under individual control — Example

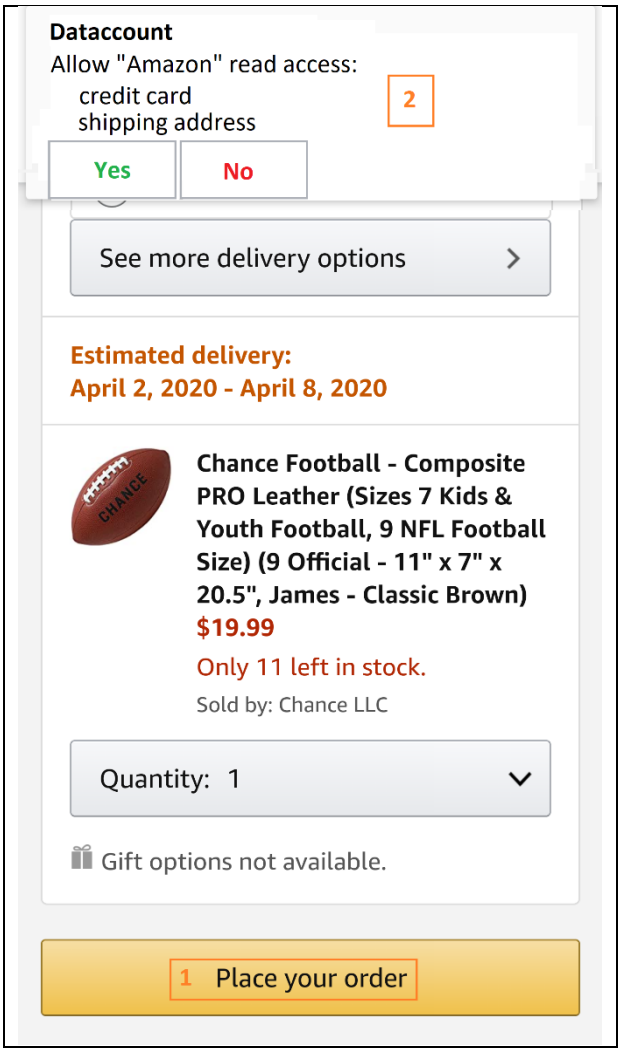
When I order something on Amazon today, they show me a screen that contains my address and credit card, as shown on the left. This illustrates how Amazon maintains my personal information in their private database.

The image on the right shows the same screen after Amazon has upgraded to use this system. All personal information is gone. Instead there is only a web address for a “dataccount” (“data” + “account”). This illustrates how Amazon doesn’t keep my personal information. They get it from my dataccount, which is at “https://merrittmarcthompson-dataccount”:





When I click the Amazon “Place your order” button (1 below), Amazon tries to access my dataccount. That causes a push notification from the Dataccount software on my phone (2), which lets me approve the access. This allows Amazon to bill my credit card and print a shipping label without permanently storing my personal data:



B. Tracking access to personal information

When a company accesses my data, my dataccount software logs the access and its purpose. You can view the accesses in the dataccount management web UI.

Access date/time	Accessor	Data Item	Access	Allowed?	Purpose
4/7/2025 19:03:27	www.amazon.com	credit card	Read	Yes	Purchase $19.99 football
4/7/2025 19:03:27	www.amazon.com	shipping address	Read	Yes	Print delivery label

C. Restricting storage of personal information

Thus far, this system gives external entities a way to access necessary information without storing it, but it doesn't stop them from doing so anyway, i.e. they can still access the information from the dataccount, then also save it privately for their own use or profit.

This would be prevented by a new law that would criminalize storage of other people's personal data for purposes other than the stated ones.

In the example above, if Amazon got your shipping address, then used it for any purpose other than printing a delivery label (for example, if they sold it to a mass-mailing company), they would have committed a crime.

The key concept

This law would criminalize what is now an extremely common practice. You might think that companies would oppose this restriction, but it is more likely that they would ultimately approve of it, because as long as they follow the rules, they are completely absolved of liability for individuals' data losses.

Thus, this isn't a magical technological system for tracking information across the web. Instead, the goal of the technology is simply to make it possible to create a new social norm.

About compensation

In this scheme, your Roomba vacuum could make a floor plan of your house, then send it directly to your dataccount (sending it anywhere else would be illegal). Roomba could then buy your floor plan from you. I'm not sure if that would make you much money, but Roomba might give customers perks of some kind for it. You could also sell it to researchers other than Roomba, since it is your data, not Roomba's.

Another example like this would be selling medical data to researchers. If you have a rare or interesting condition, you might get a reasonable amount of money for your medical records.

Implementation overview

Dataccount location

A dataccount can go on any server that is on the web, so long it supports the dataccount web API. Thus, you could have your own server locked in a closet in your house. Or you could contract with a "data bank" company that provides encrypted dataccount services for many individuals.

Data format

Conceptually, personal data would be stored in a *data items* table in the form of key/value pairs, for example:

> "Home address"="Jane Smith, 123 Shady Lane, Tulsa, Oklahoma, 74101"

Nondisclosure of data in keys

Imagine that you want to send a diamond ring to your mistress Sheila's home address. Further, you already have a data key called "My mistress Sheila's home address". So, you might fill out a field like this on the Amazon web site:

> Shipping Address Data Key: `My mistress Sheila's home address`

This would be less than ideal, as it would disclose personal information. To prevent this, you could create an *alias* in your dataccount:

> "Amazon shipping address"=@"My mistress Sheila's home address"

This means, “Whenever someone asks for ‘Amazon shipping address’, give them ‘My mistress Sheila’s home address’ instead”.

Amazon would default their form to the following, so you probably wouldn’t have to enter anything:

> Shipping Address Data Key: `Amazon shipping address`

APIs

There would be two API sets:

- One API set is for individuals (“owners”) to manage their own personal information. This includes functions that support the on-the-fly push notifications for Allow and Deny illustrated previously.
- One API set is for companies (“accessors”) to access personal information of others. This is just two functions, one to read and one to write.

Owner authentication

Owners can be authenticated for the owner API using HTTP Basic authentication or OAuth. Encrypted HTTPS transport must be used for HTTP Basic to work, as the password is merely base-64 encoded.

Accessor authentication

The most important thing for the accessor API is to authenticate the accessor. For example, if someone tried to access your dataccount, said they were “Amazon”, and the dataccount simply believed it without authentication, anyone could impersonate Amazon and get your information.

Transport Layer Security (TLS), the basis of the HTTPS communication protocol, normally requires the server to prove who they are to the client before any data can pass between them. The server does this by using a secret private key to encrypt some data. If the client can successfully decrypt the data using the public key, it proves to the client that the server is who they claim to be.

But that process is the opposite of what this system requires. In this system the server (the dataccount) must obtain proof that the client (the accessor, ex. Amazon) is really who they claim to be, not the other way around.

Fortunately, there is a “two-way” variant of TLS which provides exactly this function. It both verifies that the server is who they claim to be to the client (as normal) and the client is who they claim to be to the server (needed for this system).

Push notifications and state-free web API flow

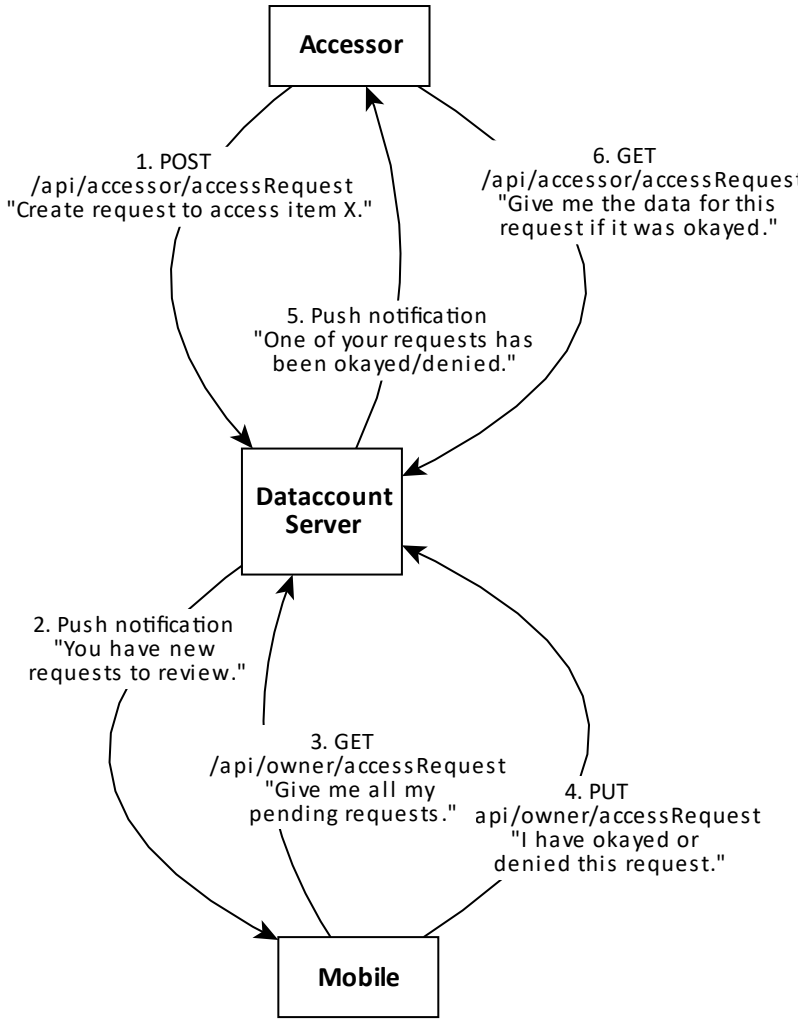
Ideally, when a company contacts your dataccount web API to access data, your server should immediately notify your mobile device for approval. It’s possible to set up a “service worker” on the mobile device via JavaScript, which always runs and waits for such notifications.

But even so, the system is designed to not rely on the push notifications. A typical use case is like this, with Amazon as an example:

1. Amazon wants to get your shipping address, so it requests it from your server. Then it continues about its business. It doesn’t wait to get the data. It might take any amount of time to get the okay from you.
2. Next, the server may or may not be able to make a push notification to your phone, telling you that you have something to approve.
3. At some point, because you saw a notification or because you happened to check your pending requests, you see Amazon’s request and okay it.
4. The server may or may not be able to send a push notification to Amazon, telling them that they have an okay.
5. At some point, because of a notification or because they check periodically, Amazon requests the address.

Here are some more details of how the APIs work in the ideal case. In typical REST API form, everything is mediated by a resource, in this case, the Access Requests table. Robust REST design is good database design. Since all state is stored in the table, there can be a server crash right after step 1 (POST) or step 4

(PUT) and when the server restarts, the system will carry on based on the table data as though nothing happened:



Automatic approval or denial

This design doesn't go into the details, but there is also an automatic approval option. This allows the owner to specify that certain accessors can automatically access certain data items without manual owner approval, because the owner trusts the accessor and is tired of manually approving their requests. For example, you could fill in a form like this in the owner UI:

Automatic approval

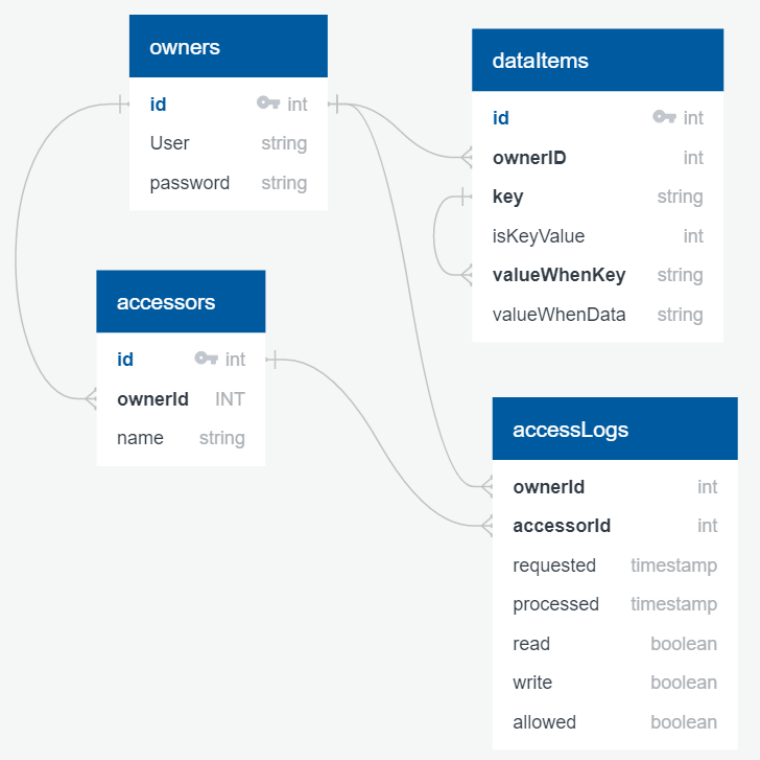
For accessor: www.amazon.com

Data items: Amazon Shipping Address
Amazon Credit Card Number

Restrictions: Up to 5 accesses per month

Database schema

The database looks like this (this is work in progress and is somewhat out of sync with the previous notes):



Here's a preliminary database schema in MySQL:

```
DROP DATABASE dataccount;
CREATE DATABASE dataccount;
USE dataccount;

CREATE TABLE `owners`
(
  `id` INT AUTO_INCREMENT PRIMARY KEY,
  `user` VARCHAR(32) NOT NULL UNIQUE,
    -- This is an ID created by the user to identify himself.
    -- It must be unique to the dataccount server. It could be "bob", "mthompso913", etc.
  `password` VARCHAR(32)
    -- Used for HTTP Basic authentication over HTTPS.
);

INSERT INTO `owners` (`user`, `password`) VALUES ('example user', 'example password');
SELECT `id` INTO @exampleOwnerId FROM `owners` WHERE user='example user';

CREATE TABLE `dataItems`
(
  `id` INT AUTO_INCREMENT PRIMARY KEY,
```

```
  `ownerId` INT,
    -- Which owner owns this item.
  CONSTRAINT fk_dataItem_ownerId FOREIGN KEY (`ownerId`) REFERENCES `owners`(`id`),
  `key` VARCHAR(64) NOT NULL UNIQUE,
    -- A unique string that identifes the item, ex. "Amazon shipping address",
    -- or "Bob's credit card number". The program may normalize this by removing extra spaces.
  `isKeyValue` BOOLEAN NOT NULL,
    -- TRUE if the valueWhenKey is used. FALSE if the valueWhenData is used.
  `valueWhenKey` VARCHAR(64),
  CONSTRAINT fk_dataItem_valueWhenKey FOREIGN KEY (`valueWhenKey`) REFERENCES `dataItems`(`key
`),
  `valueWhenData` VARCHAR(255)
    -- The value of the data item is either a key to another data item (valueWhenKey),
    -- or the final data itself (valueWhenData).
);

INSERT INTO dataItems (`ownerId`, `key`, `isKeyValue`, `valueWhenData`, `valueWhenKey`)
  VALUES (@exampleOwnerId, 'example key', 0, 'example data', NULL);

CREATE TABLE `accessors`
(
  `id` INT AUTO_INCREMENT PRIMARY KEY,
  `ownerId` INT,
  CONSTRAINT fk_accessor_ownerId FOREIGN KEY (`ownerId`) REFERENCES `owners`(`id`),
    -- Every owner in the server has his own, private list of accessors.
    -- Ex. "www.amazon.com" for me is different from the one for you.
    -- This is just for privacy: other owners can't find out who you are dealing with.
  `name` VARCHAR(64)
    -- Ex. "www.amazon.com".
);

CREATE TABLE `accessLogs`
(
  `ownerId` INT,
    -- What owner owns the log.
  CONSTRAINT fk_accessLog_ownerId FOREIGN KEY (`ownerId`) REFERENCES `owners`(`id`),
  `accessorId` INT,
    -- Which accessor tried to do the access.
  CONSTRAINT fk_accessLog_accessorId FOREIGN KEY (`accessorId`) REFERENCES `accessors`(`id`),
  `requested` TIMESTAMP NOT NULL,
    -- The time the access was requested by the accessor.
  `processed` TIMESTAMP,
    -- The time the access was allowed or denied by the owner. NULL when pending.
  `read` BOOLEAN NOT NULL,
    -- Whether the access was for read.
  `write` BOOLEAN NOT NULL,
    -- Whether the access was for write.
  `allowed` BOOLEAN
```

```
    -- Was the access allowed? NULL when pending.
);
```

Owner web API

Authentication and authorization

Every owner web API function requires this standard HTTP Basic Authorization header (a description of the OAuth mechanism is pending):

```
Authorization: Basic <base-64 value>
```

The base-64 value decodes to "<user>:<password>". If the user and password are in the Owners table, use of the function is authorized. For example, the "get data items" API on the server could use this statement to both access the owner's data item values and authorize the operation:

```
SELECT dataItems.key, dataItems.isKeyValue, dataItems.valueWhenKey, dataItems.valueWhenData
FROM dataItems, owners
WHERE
  owners.`user` = ? AND
  owners.`password` = ? AND
  dataItems.ownerId = owners.id
```

Errors:

- 400 Bad Request: Missing or incorrect Authorization header.
- 401 Unauthorized: Any SQL error.

GET /api/owner/dataItems

Returns an application/json value, ex.:

```
[
  ["key":"example key", "isKeyValue":"0", "valueWhenKey":"", "valueWhenData":"example data"],
  ["key":"example key", "isKeyValue":"1", "valueWhenKey":"other key", "valueWhenData":""]
]
```

POST /api/owner/dataItems

The HTTP body is a JSON list of new items to add in the same format as the data items GET function above.

GET /api/owner/accessLogs

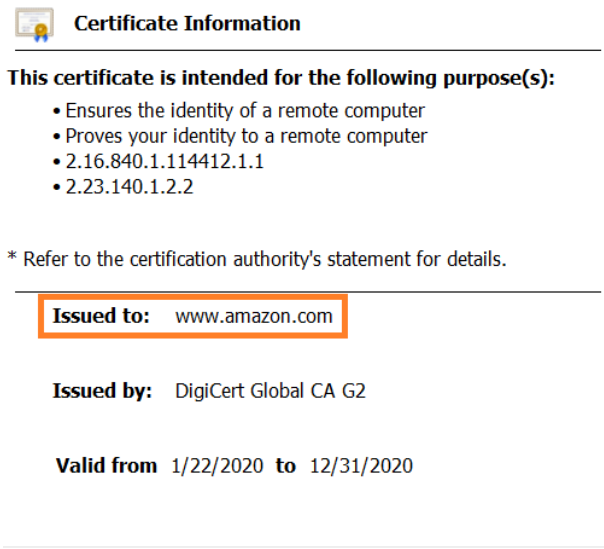
[more APIs in progress]

Accessor web API

Authentication

Authentication is provided by two-way TLS. Two-way TLS lets the server (the dataccount) get and verify the client's certificate. The "Issued to:" value in the certificate becomes the "name" value in the

dataccount accessors table. As an example, here's the TLS certificate for Amazon. The accessors.name value would be "www.amazon.com":



Because TLS does the verification, the server only receives verified requests and needs to do no further validation. Whenever an Accessor API message is received, this happens on the server:

1. If the TLS "Issued To" value is not in any accessors table name column, the server adds a new accessor row to the table for it. This lets the owner see a list of all accessors.
2. The server adds an accessLogs row for the access. This lets the owner see a list of all attempted accesses.

Thus, the "data items" access functions actually touch three tables: accessors, accesslogs, and dataItems.

The API contains two functions:

GET /api/accessor/dataItems
The body would be this JSON:

```
{
  "owner":"<owner ID>",
  "key":"<data item key>,
  "purpose:"<purpose description text>"
}
```

The result is one of these:

- 400 Bad Request: Something was wrong with the JSON request body.
- 403 Forbidden: The owner denied the request.
- 200 OK: The owner allowed the request. The data is in the result body.

PUT /api/accessor/dataItems
The body would be this JSON:

```
{
  "owner":"<owner ID>",
  "key":"<data item key>,
  "purpose:"<purpose description text>",
  "value":"<new value>"
}
```

The result is one of these:

- 400 Bad Request: Something was wrong with the JSON request body.
- 403 Forbidden: The owner denied the request.
- 200 OK: The owner allowed the request.