What's Fresh Documentation

Release 1.0

Megan Goossens, Evan Tschuy

August 26, 2015

Contents

API and Usage Documentation:

CHAPTER 1

Usage

The What's Fresh API web app is used to enter the data that is displayed in the What's Fresh Mobile app. It consists of a series of simple forms for each type of object that can be added to the database. Required fields are marked with a *, and any errors or missing items will be flagged when the form is saved. The data will not be saved until errors are corrected.

1.1 Objects

There are six major objects in What's Fresh: Vendors, Products, Preparations, Stories, Videos and Images. When you log in to the application, the first screen you see is the Entry screen, which lists the objects. Clicking on one of the object buttons will bring you to a screen listing all of the objects already saved in the system.

For example, clicking Vendors will display a table listing all the Vendors currently in the system. On this screen you can add a new Vendor by clicking the yellow "New Vendor" button at the top of the list, or edit any of the existing Vendors by clicking on that Vendor.

1.2 Adding and Editing

The forms for editing and for adding new objects are the same, except that the edit form will already be filled out with the existing data. You can edit this data and save the changes using the "Save" button at the bottom of the form, or delete the entire record using the red "Delete" button on the top of the form.

1.2.1 Workflow

The objects in What's Fresh sometimes depend on other objects. For example, Vendors need Products, so you can't add a new Vendor without adding Products first. We recommend the following work flows to add different objects:

Products

1. Determine what Preparations are available for this Product (smoked, dried, fresh, etc).
2. Create the Preparation objects if they don't already exist.
3. If there is an Image for this Product, create an Image object (be sure to give the image a unique and descriptive name).
4. If this Product has a Story, make sure that Story exists (for instance, the Salmon story will probably be shared by all varieties of Salmon).
5. Create the Product object, selecting the correct Story and Image, and add each applicable preparation.

Vendors

1. Make sure this Vendor's Products are added (see above).
2. If the Vendor has a Story (rare), create the Story object.
3. Create the Vendor object, adding the correct Products and selecting a Story, if applicable.

Stories

1. If this Story includes Images, create the Images.
2. If this Story includes Videos, create the Videos.
3. Create the Story object, adding the correct Images and Videos, if applicable.

See below for details on adding these objects.

1.2.2 Vendors

These are the records for businesses that sell products. Vendors are also specific to a location, so if Bob's Fish has two different locations where they sell their Products, each location will be a separate Vendor. These can be distinguished by name, for instance Bob's Fish Newport and Bob's Fish Waldport.

The address for a Vendor should be the actual location where they sell their Products, not an office or P.O. box.

Pre-requisites

Vendors sell Products, so in order to create a new Vendor, some Products must exist (ideally, the specific Products that Vendor sells). Before creating a new Vendor, it is a good idea to make sure their Products exist. The Vendor form *requires* at least one Product to be added to the Vendor. A Vendor's Product list can be changed later.

Required Data

Certain information is required to create a new Vendor, make sure you know these items before starting:

Name The name of the business.

Hours The typical hours of operation.

Description A brief description of the business.

Address

The street address where the products are being sold.

- Street Address
- City
- State
- Zipcode

Contact Name The primary contact name for this Vendor.

Products (At least one product should be added).

Note: When a Product is added, you must also select a Preparation for that product. A vendor may sell different preparations for the same Product, or only one of many possible preparations for a Product. For example, a Vendor may sell smoked, frozen and fresh salmon, and also may sell shrimp, but only frozen, not fresh. Every Product/Preparation combination the vendor sells should be added.

Note: Street addresses are turned into GPS coordinates for display on a map in the Mobile app, so it is important to be accurate.

Optional Data

Additionally, there are several optional fields:

Story Select from an existing Story (see the entry on Story objects below)

In Port The current status of the Vendor, if they sell from a boat, or only when the boat is in port. (Not used currently).

Location Description Additional details about how to find the Vendor location (The red boat at the end of Dock 3, for example).

Website The Vendor's website.

Email The Vendor's primary email address.

Phone The Vendor's phone number..

1.2.3 Preparations

Preparations are the way in which a Product can be prepared for sale. This can include fresh, frozen, live, smoked, cooked, dried, and many more.

Pre-requisites

Preparations have no prerequisites.

Required Data

Preparations require the following fields to be filled out:

Name The name of this Preparation.

Optional Data

These fields are optional:

Description A more detailed description of the preparation. For instance 'Fermented' might require a little more explanation than 'Frozen'.

Additional Information Use this field to note additional things a user might need to know about buying Products with this Preparation. For example, fresh fish should be kept in a cooler for a long ride home.

1.2.4 Products

Products are what Vendors sell, and the central Object in What's Fresh.

Note: Different varieties of a particular product should be treated as separate products, if they are sold as such. For instance, different varieties of Salmon are sold with different prices, therefore Coho, Chinook and Sockeye salmon should be separate products. The 'Specific Variety' field of all these Products will be 'Salmon', and each will have a different value in the 'Product Name' field.

Pre-requisites

Products require Preparations. Make sure all the possible preparations this Product can have are created first. If an Image or Story is going to be added, these objects should be created before adding the Product.

Required Data

Products require the following fields to be filled out:

Specific Variety The common name of this Product (i.e. Salmon).

Description A brief description of the product.

Season The typical season for this Product (ex. 'Sept. 20 - Dec 20', or 'Spring and Fall').

Market Price The current market price for this Product.

Preparation At least one preparation must be added.

Optional Data

These fields are optional:

Product Name The name of this product (ex. Coho, Sockeye, etc).

Alternate Name Other name(s) this product might be commonly called.

Origin The geographic origin of this Product.

Available Indicate if this product is currently being sold (ex. a fish is available even though its normal season is over).

Link A link to an official web site for this Product (ex. National Shrimp Council website).

Image A representative image of this Product.

Story The Story of this Product (see Stories below).

1.2.5 Stories

Stories are collections of educational information about a Product or Vendor. Stories may be shared by many varieties of a particular Product - for instance the Salmon Story will likely apply to Coho, Chinook, and Sockeye salmon, which are all distinct Products.

Pre-requisites

If Images or Videos are going to be added to this Story, they should be created before the Story is created.

Required Data

Stories require the following fields to be filled out:

Name A name for this story. (This should be unique and easy to identify from the Story pull-down menu on the Product and Vendor forms.)

Optional Data

Facts A list of facts about the Product or Vendor.

History Text about the history and historical importance of the Product or Vendor.

Buying (Products only) What to know about buying this Product, (for example: how to select for freshness and quality).

Preparing (Products only) Ways to prepare this Product, recipes and other tips.

Products (Product only) Derivative Products made from this Product.

Season (Product only) Detailed information about the season for this Product.

Images One or more images related to this Product.

Videos One or more videos related to this Product.

1.2.6 Videos

Videos are external links to videos hosted on YouTube, Vimeo, or elsewhere. Any video that can be streamed can be used here.

Pre-requisites

Videos have no pre-requisites.

Required Data

Videos require the following fields to be filled out:

Name A name for this Video. (This should be unique and easy to identify from the Video pull-down menu on the Story form.)

Link The URL for this video (ex. https://www.youtube.com/watch?v=hl3wWwouOUE).

Caption A brief descriptive caption for this Video.

Optional Data

Videos have no optional fields.

1.2.7 Images

Images are uploaded image files. The Image upload form accepts .jpg, .png, and .gif image files. Images may be displayed as a single representative image for a Product in a Product view, or as part of a slideshow of images in a Story.

Pre-requisites

Images have no pre-requisites.

Required Data

Images require the following fields to be filled out:

Image Upload an image file.

Name A name for this Image. (This should be unique and easy to identify from the Image pull-down menu on the Story and Product forms.)

Caption A brief descriptive caption for this Image.

Optional Data

Images have no optional fields.

CHAPTER 2

Admin

2.1 Adding New Users

Adding a new user is simple. First, log into /admin. Under Authentication and Authorization there is Groups and Users. To the right of these are two buttons that say Add and Change. Click on Add. Enter the user's username and password. Click save in the bottom right hand corner. You will be taken to a Change User page. Here you can edit the user's information. They have already been added to the Data Entry Users group. This is to prevent an infinite loop when they log in. Once you are done, click save at the bottom of the page. Congratulations, you've just added a user!

CHAPTER 3

API Endpoints

The What's Fresh API is a REST-style JSON API for discovering fresh local food products.

The data can be accessed through a handful of endpoints:

Endpoints

Every API return will include an `error` hash, containing an error status, error name, error text, and error level. If the error status is True, the data should be considered bad and ignored. The error name will return a human-readable error name, like "Product Not Found", and the error text will contain slightly more detail, including the ID of the object not found.

The API will only return HTTP 200 status codes, including for errors, except in the case of server-side errors, which will return a 500.

If future additions are made to the API, they will be made in the `ext` extension dictionary so as to provide backward compatibility.

3.1 Products listing

The products listing is available at `/products/`. It returns a JSON array consisting of each of the products, and information about them.

3.1.1 Optional Fields

The following fields in a product can be either a value, or null:

- variety: text or empty string
- alt_name: text or empty string

- origin: text or empty string
- link: valid URL or empty string
- available: boolean or null

3.1.2 Parameters

The `/products/` endpoint accepts the `limit=<int>` parameter, limiting the number of products returned to the number requested. For instance, `/products?limit=5` will limit the number of results returned to 5.

3.1.3 Example: GET /products/


```
{
  "error": {
    "status": false,
    "text": null,
    "name": null,
    "debug": null,
    "level": null
  },
  "products": [
    {
      "origin": "Pacific Ocean",
      "available": null,
      "description": "A classic fish",
      "variety": "Salmon",
      "season": "July - October",
      "image": null,
      "created": "2014-09-18T18:33:22.140Z",
      "modified": "2014-09-24T19:42:52.720Z",
      "market_price": "$100 per fluid ounce",
      "link": "",
      "alt_name": "Oncorynchus kisutch",
      "story_id": null,
      "id": 1,
      "name": "Coho Salmon"
    },
    {
      "origin": "Pacific Ocean",
      "available": null,
      "description": "A popular seafood prized for its sweet and tender flesh. ",
      "variety": "Dungeness",
      "season": "December to January",
      "image": null,
      "created": "2014-09-18T18:36:14.240Z",
      "modified": "2014-09-24T19:43:08.960Z",
      "market_price": "$0.10 per dozen",
      "link": "",
      "alt_name": "Metacarcinus magister",
      "story_id": null,
      "id": 2,
      "name": "Dungeness Crab"
    },
    ...
    ]
}
```

3.2 Product details

The `/products/<id>` endpoint returns the same data as `/products`, but only for the product specified by id. This is used when the ID of a product is known, but the details of the product are not – for instance, getting details on a product after finding its ID and name through vendor information.

3.2.1 Optional Fields

The following fields in a product can be either a value, or null:

- variety: text or empty string
- alt_name: text or empty string
- origin: text or empty string
- link: valid URL or empty string
- available: boolean or null

3.2.2 Example: GET /products/2


```
{
  "error": {
    "status": false,
    "debug": null,
    "text": null,
    "name": null,
    "level": null
  },
  "origin": "Pacific Ocean",
  "available": null,
  "modified": "2014-09-24T19:43:08.960Z",
  "description": "A popular seafood prized for its sweet and tender flesh. ",
  "variety": "Dungeness",
  "season": "December to ???",
  "image": null,
  "created": "2014-09-18T18:36:14.240Z",
  "market_price": "$0.10 per dozen",
  "link": "",
  "alt_name": "Metacarcinus magister",
  "story_id": null,
  "id": 2,
  "name": "Dungeness Crab"
}
```


3.3 Vendors listing

The vendors listing is available at `/vendors/`. It returns a JSON array consisting of each of the vendors, and information about them.

Note: Coordinates used in the API are standard, decimal degree coordinates. Many results will contain negative coordinates.

3.3.1 Optional Fields

The following fields in a vendor can be either a value, or null:

- status: boolean or null
- location_description: text or empty string
- phone: valid phone number (with international prefix) as string or null
- website: valid URL or empty string
- email: valid email or empty string

3.3.2 Parameters

Limit

The `/vendors/` endpoint accepts the `limit=<int>` parameter, limiting the number of vendors returned to the number requested. For instance, `/vendors?limit=5` will limit the number of results returned to 5.

Location

It also accepts `lat=<float>` and `long=<float>` parameters. When these are provided, the results will be returned sorted by proximity, with the closest vendor listed first. For instance, `/vendors?lat=44.618808&long=-124.049905` will provide results sorted by distance to the Hatfield Marine Science Center in Newport, OR. If only one of the parameters is provided, it will be ignored.

Proximity

The `proximity=<int>` parameter can be used in conjunction with the `lat` and `long` parameters. It will restrict the results to those within the given number of miles. To get a list of vendors within 10 miles of the Hatfield Marine Science Center, then, the following could be queried:

`/vendors?lat=44.618808&long=-124.049905&proximity=10`

As it requires the user's location, it will be ignored if the `lat` and `long` positions are not also provided.

3.3.3 Example: GET /vendors/


```
{
  "error": {
    "error_status": false,
    "error_name": null,
    "error_text": null,
    "error_level": null
  },
  "vendors": [
    {
      "status": null,
      "city": "Newport",
      "website": "",
      "modified": "2014-09-24T19:55:16.085Z",
      "description": "A local tuna provider.",
      "zip": "97365",
```

```
      "created": "2014-09-23T23:52:51.484Z",
      "story_id": 1,
      "ext": {
      },
      "location_description": "",
      "email": "",
      "hours": "",
      "phone": null,
      "state": "Oregon",
      "street": "1398 SW Bay St",
      "products": [
        {
          "preparation": "Filet",
          "preparation_id": 3,
          "product_id": 3,
          "name": "Albacore Tuna"
        }
      ],
      "lng": 44.6266099,
      "lat": -124.0565731,
      "contact_name": "Todd Sherman",
      "id": 2,
      "name": "Todd's Tuna Farm"
    },
    {
      "status": true,
      "city": "Gold Beach",
      "website": "",
      "modified": "2014-09-24T20:49:33.156Z",
      "description": "Best shark meat in the west.",
      "zip": "97444",
      "created": "2014-09-23T23:59:20.016Z",
      "story_id": 1,
      "ext": {
      },
      "location_description": "",
      "email": "",
      "hours": "",
      "phone": null,
      "state": "Oregon",
      "street": "29985 Harbor Way",
      "products": [
        {
          "preparation": "Live",
          "preparation_id": 1,
          "product_id": 5,
          "name": "Leopard Shark"
        }
      ],
      "lng": 42.4210811,
      "lat": -124.4179554,
      "contact_name": "James Renolds",
      "id": 3,
      "name": "The Shark Shop"
    },
    ...
  ]
}
```

3.4 Vendors selling a product

If a user wants to know which vendors are selling a given product, the `/vendors/products/<id>` endpoint should be used. This endpoint returns a list of all vendors selling the product given by the ID in the same format as the `/vendors/` endpoint.

3.4.1 Optional Fields

The following fields in a vendor can be either a value, or null:

- status: boolean or null
- location_description: text or empty string
- phone: valid phone number (with international prefix) as string or null
- website: valid URL or empty string
- email: valid email or empty string

3.4.2 Parameters

Limit

The `/vendors/products` endpoint accepts the `limit` parameter, limiting the number of vendors returned to the number requested. For instance, `/vendors/products/3?limit=5` will limit the number of results returned to 5.

Location

It also accepts `lat=<float>` and `long=<float>` parameters. When these are provided, the results will be returned sorted by proximity, with the closest vendor listed first. For instance, `/vendors/products/3?lat=44.618808&long=-124.049905` will provide results sorted by distance to the Hatfield Marine Science Center in Newport, OR. If only one of the parameters is provided, it will be ignored.

Proximity

The `proximity=<int>` parameter can be used in conjunction with the `lat` and `long` parameters. It will restrict the results to those within the given number of miles. To get a list of vendors selling the product with ID #3 within 10 miles of the Hatfield Marine Science Center, the following could be queried:

`/vendors/products/3?lat=44.618808&long=-124.049905&proximity=10`

As it requires the user's location, it will be ignored if the `lat` and `long` positions are not also provided.

3.4.3 Example: GET /vendors/products/3


```
{
  "error": {
    "error_status": false,
    "error_name": null,
    "error_text": null,
    "error_level": null
```

```
  },
{
  "vendors": [
    {
      "status": null,
      "city": "Newport",
      "website": "",
      "modified": "2014-09-24T19:55:16.085Z",
      "description": "A local tuna provider.",
      "zip": "97365",
      "created": "2014-09-23T23:52:51.484Z",
      "story_id": 1,
      "ext": {
      },
      "location_description": "",
      "email": "",
      "hours": "",
      "phone": null,
      "state": "Oregon",
      "street": "1398 SW Bay St",
      "products": [
        {
          "preparation": "Filet",
          "preparation_id": 3,
          "product_id": 3,
          "name": "Albacore Tuna"
        }
      ],
      "lng": 44.6266099,
      "lat": -124.0565731,
      "contact_name": "Todd Sherman",
      "id": 2,
      "name": "Todd's Tuna Farm"
    },
    {
      "status": null,
      "city": "Waldport",
      "website": "",
      "modified": "2014-09-24T20:50:37.652Z",
      "description": "The freshest seafood in Waldport.",
      "zip": "97394",
      "created": "2014-09-24T00:06:43.426Z",
      "story_id": 1,
      "ext": {
      },
      "location_description": "",
      "email": "",
      "hours": "",
      "phone": null,
      "state": "Oregon",
      "street": "98 NW Alsea Bay Dr",
      "products": [
        {
          "preparation": "Live",
          "preparation_id": 1,
          "product_id": 7,
          "name": "Savory Clam"
        },
```

```
        {
          "preparation": "Filet",
          "preparation_id": 3,
          "product_id": 3,
          "name": "Albacore Tuna"
        }
      ],
      "lng": 44.4269468,
      "lat": -124.0792542,
      "contact_name": "Carlos Molena",
      "id": 4,
      "name": "Waterfront Seafood Shop"
    }
    ]
}
```


3.5 Vendor details

The `/vendors/<id>` endpoint returns the same data as `/vendors`, but only for the vendor specified by id. This is used when the ID of a vendor is known, but the details of the vendor are not – for instance, getting details on a vendor after finding its ID and name through the vendors-for-product list.

3.5.1 Optional Fields

The following fields in a vendor can be either a value, or null:

- status: boolean or null
- location_description: text or empty string
- phone: valid phone number (with international prefix) as string or null
- website: valid URL or empty string
- email: valid email or empty string

3.5.2 Example: GET /vendors/2


```
{
  "error": {
    "debug": null,
    "status": false,
    "text": null,
    "name": null,
    "level": null
  },
  "website": "",
  "street": "1398 SW Bay St",
  "lng": 44.6266099,
  "contact_name": "Todd Sherman",
  "city": "Newport",
  "zip": "97365",
  "story_id": 1,
  "location_description": "",
  "id": 2,
```

```
  "state": "Oregon",
  "email": "",
  "status": null,
  "modified": "2014-08-08T23:27:05.568Z",
  "description": "A local tuna provider.",
  "hours": "",
  "phone": null,
  "lat": -124.0565731,
  "name": "Todd's Tuna Farm",
  "created": "2014-08-08T23:27:05.568Z",
  "ext": {
  },
  "products": [
    {
      "preparation": "Filet",
      "preparation_id": 3,
      "product_id": 3,
      "name": "Albacore Tuna"
    }
  ]
}
```


3.6 Story details

The `/stories/<id>` endpoint returns the story for a given ID.

3.6.1 Example: GET /stories/2


```
{
  "error": {
    "error_status": false,
    "error_name": null,
    "error_text": null,
    "error_level": null
  },
  "story": "A story can contain various bits of text."
}
```


3.7 Preparation details

The `/preparations/<id>` endpoint returns the preparation details for a given preparation ID.

3.7.1 Example: GET /preparations/4


```
{
  "error": {
    "status": false,
    "debug": null,
    "text": null,
    "name": null,
```

```
    "level": null
  },
  "name": "Smoked",
  "description": "Thats dense stuff, tastes like forest fire.",
  "additional_info": "",
  "id": 4
}
```


3.8 Locations list

The `/locations/` endpoint returns a list of all the cities this vendors are in. Each city is given an location index, and a name. The index is not guaranteed to stay the same.

3.8.1 Example: GET /locations/


```
{
  "error": {
    "status": false,
    "name": null,
    "text": null,
    "debug": null,
    "level": null
  },
  "locations": [
    {
      "location": 1,
      "name": "Gold Beach"
    },
    {
      "location": 2,
      "name": "Corvallis"
    },
    {
      "location": 3,
      "name": "Florence"
    },
    {
      "location": 4,
      "name": "Newport"
    }
  ]
}
```

CHAPTER 4

Install

CHAPTER 5

Planning

5.1 Draft API

5.1.1 Format

Responses will be returned in standard JSON format. An attempt will be made to keep the structure simple. Https will be used for all endpoints.

Null values (optional fields that do not have data), will be empty strings: “”.

5.1.2 Versions

The API will be versioned with simple version integers, 1, 2, 3, ...

ex: https://whatsfresh.org/1/vendors

5.1.3 Errors

Error records will be returned in every message, and will consist of a dictionary containing the error status, error name, error text and error level. The status field will indicate the presence of an error condition, and should be checked before attempting to process the rest of the response.

example:

```
error: {error_status: true, error_name: 'not_found_error', error_text: 'product with id=232 could not
```

5.1.4 Extended Fields

To allow for future expandability, a dictionary call ‘ext’ will be included with every response. This dictionary will either contain no records, or will contain additional first-class records that were not included in the original specification. For instance, if a new attribute “color” is later added to the product response, it can be included in the extended attributes array. Applications can choose to discover/use these new fields or ignore them without effecting backwards compatibility. Response validation should include the presence of ext, but not its contents.

5.1.5 Endpoints

/locations

Return a list of city names, such as Newport, Florence, Waldport, etc.

```
{
        locations: [
            'Newport',
            'Florence',
            'Astoria'
        ]
}
```

/products

Return a dictionary containing a record for every product in the database.The product id is the record key. This data is unlikely to change frequently, it should be in long-term storage on the device and refreshed periodically.

```
{
        error: {error_status: bool, error_name: text, error_text: text, error_level},
    <product_id>: {
                name: text
                variety: text or null
                alt_name: text or null
                description: text
                origin: text or null
                season: text
                available: bool or null
                market_price: text
                preparations: [smoked, fresh, live...]
                link: url or null
                image: int or null
                story: int or null
                created: datetime
                modified: datetime
                ext: {attribute: value, attribute: value...} or {}
        },
        <product_id>: {...},
        ...
}
```

/products/<id>

Returns a single product record identified by <id>. This may be useful for selectively refreshing the local master list of products fetched by /products.

```
{
        error: {error_status: bool, error_name: text, error_text: text, error_level},
        id: int
        name: text
        variety: text or null
        alt_name: text or null
        description: text
        origin: text or null
        season: text
        available: bool or null
        bool: bool
        market_price: text
        preparations: [text, text, text...]
```

```
        link: url or null
        image: int or null
        story: int or null
        created: datetime
        modified: datetime
        ext: {attribute: value, attribute: value...}
}
```

/products/describe

Returns a description of the fields in a product record. These should correspond to internal docstrings, which in turn should be extracted into the master project documentation.

```
{
        endpoint_description: "text describing the endpoint"
        id: "text describing this field"
        name: "text describing this field"
        ...
}
```

/vendors

Return a dictionary containing a record for each vendor in the database. The vendor id is the record key. Each vendor record also contains a dictionary of products carried by this vendor. This data is likely to change more often, and should be cached locally but refreshed for specific products or locations whenever possible.

```
{
        error: {error_status: bool, error_name: text, error_text: text, error_level},
        <vendor_id>: {
                name: text
                status: bool or null
                description: text
                lat: float
                long float
                street: text
                city: text
                state: text
                zip: text
                location_description: text or null
                contact_name: text
                phone: text or null
                website: url or null
                email: email or null
                story: int or null
                ext: {attribute: value, attribute: value...}
                created: datetime
                updated: datetime
                products: {
                        <product_id>: {name: text, preparation: text},
                        <product_id>: {name: text, preparation: text},...
                }
        },
        <vendor_id>: {...},
        ...
}
```

/vendors/<id>

Returns a single vendor record identified by <id>. This should be used to fetch data whenever a specific vendor id is known.

```
{
        id: int
        name: text
        status: bool or null
        description: text
        gps_location: coords
        street: text
        city: text
        state: text
        zip: text
        location_description: text
        contact_name: text
        phone: text or null
        website: url or null
        email: email or null
        story: int or null
        ext: {attribute: value, attribute: value...}
        created: datetime
        updated: datetime
        products: {
                <product_id>: {name: text, preparation: text},
                <product_id>: {name: text, preparation: text},...
        }
}
```

/vendors/describe

Returns a description of the fields in a vendor record. These should correspond to internal docstrings, which in turn should be extracted into the master project documentation.

```
{
        id: (text describing this field)
        ...
}
```

/stories/<id>

Returns a story record identified by <id>.

```
{
name: text
history: text
facts: text
buying: text
preparing: text
products: text
season: text
images: [
    {name: text, caption: text, link: text}
    {name: text, caption: text, link: text}
    ...
]
videos: [
    {name: text, caption: text, link: url}
    {name: text, caption: text, link: url}
    ...
]

}
```

/images/<id>

Returns an image record identified by <id>. Alternatively, this could return the image data itself as content-type image rather than json.

```
{
        image: "url to image"
        caption: "text" or null
        name: text
}
```

/vendors/product/<id>

Returns a dictionary of vendors that carry a product identified by <id>. The records are identical to those returned by /vendors/<id>, but filtered by the product id.

/nearby/?lat=<float>&long=<float>

Returns nearby available vendors. Vendor records are as defined above, including the products array.

5.1.6 Additional parameters

These parameters can be added to any endpoint request

?location=<lat>,<long>

or

?lat=<float>&long=<float>

These parameters represent the latitude and longitude of either the mobile device's current location, or a pre-defined location such as "Newport, OR". These will cause the results to be sorted by proximity, closest items first. This parameter will be ignored with the /stories endpoint. Depending on how the device handles the coordinates, it may be more convenient to send a single parameter, 'location=<lat>,<long>' and use the latitude and longitude as positional arguments.

examples:

?limit=<int>

This parameter will limit the number of records returned to <int>. In combination with the location parameter, it can be used to return the 5 nearest vendors selling tuna:

?proximity=<int>

This parameter will restrict the returned results to those within <int> miles (or configurable distance unit) of the given location. Ignored if no location is given.

5.2 Draft Data model

5.2.1 products

```
id              int (pk)
name            varchar
variety         varchar (optional)
alt_name        varchar (optional)
description     text
origin          varchar? (optional)
season          varchar (string describing season?)
```

```
available       bool (optional, is or is not available now?)
market_price    varchar
link            url (optional, link to industry info site)
image_id        int (optional, image foreign key)
stories_id      int (optional, image foreign key)
created         datetime
modified        datetime (auto-update on modification)
```

5.2.2 vendors

```
id                      int (pk)
name                    varchar
status                  bool (optional, in or out)
description             text
lat                     float
long                    float
street                  varchar
city                    varchar
state                   varchar
zip                     varchar
location_description    text (optional)
contact_name            varchar
phone                   varchar (optional)
website                 url (optional)
email                   email (optional)
stories_id              int (optional, story foreign key)
created                 datetime
updated                 datetime (auto-update on modification)
```

5.2.3 stories

```
    id          int (pk)
name        varchar
history     text
facts       text
buying      text
preparing   text
products    text
season      text
    created     datetime
    updated     datetime (auto-update on modification)
```

5.2.4 images

```
id          int (pk)
image       image (file)
caption     text (optional)
name        text
created     datetime
updated     datetime (auto-update on modification)
```

5.2.5 videos

```
id          int (pk)
video       url
name       text
caption     text (optional)
created     datetime
updated     datetime (auto-update on modification)
```

5.2.6 preparations

```
id                  int (pk)
name                varchar
description         text (optional)
additional_info     text (optional)
```

5.2.7 products_preparations

```
product_id          int (foreign key to product)
preparation_id      int (foreign key to preparation)
```

5.2.8 vendors_products

```
vendors_id           int (vendors foreign key)
products_id          int (products foreign key)
preparation_id       int (preparation foreign key)
vendor_price         varchar (optional)
available            bool (optional, has this product right now?)
```

5.3 Draft Error Handling

Proposal for error types and levels to be returned by API endpoints

5.3.1 The error array

```
error:     {
    status: true,
    name: 'name_of_error',
    text: 'end-user friendly error message',
    level: 'error_level'
    debug: 'detailed debug info'
}
```

5.3.2 Error Levels

Information Additional information (deprecation warning?) is available, otherwise API response is complete and correct. Information type errors should return a 200 response.

Warning A warning about the state of the database or data contained is available, API response is complete, but may be incorrect, or is correct but may be incomplete. A known error occurred, which is reported in the error text. Warnings will most likely return a 400 response.

Error Some fatal error has occurred on the API back end, and no API response can be returned, other than the error array. Errors should return a 404 or 500 response.

Error status will be true for any error level, consumer code should check both the status and level to determine the appropriate action.

5.3.3 Error names and messages

(this list will expand as we discover new ways to break things)

Object Not Found

(single object) "<object> id <requested_id> was not found." (ex: Vendor id 237 was not found)

(list) "No <object>s were found." (ex: No Vendors were found)

Malformed/bad parameter

(location) "There was an error with the given coordinates <lat, long>" debug message: "<error returned by geodjango>"

(this can be changed as we implement move form validation and catch the error before geodjango does. This can be a Warning error, as we can return some default set of data, but it won't be what the client really wanted. It may also be a 404 error, we need to determine whether to return a default data set, or nothing.)

(proximity) "Proximity <proximity> is out of range. Falling back to default <default proximity>" "Proximity <proximity> is malformed. Falling back to default <default proximity>"

(depending on the actual issue - this is a good example of the Information error level, as we default to 20 miles if the parameter is missing or bad)

(limit) "Limit <limit> is out of range. Returning all results." "Limit <limit> is malformed. Returning all results." (depending on the actual issue - this is a good example of the Information error level, as we default to ignoring the limit if it is malformed.)

5.3.4 HTTP Response codes

Responses containing specific types of errors will report an appropriate HTTP response code as well as the error array containing information about the error.

Malformed parameters: 400 Ex. coordinates given are in the wrong format

Object Not Found: 404 Ex. can't find the vendor or product requested

Code or query execution error: 500 Ex. we have an error in the code that raised an exception

An empty list of objects will return a 200 code.

Developer Setup:

CHAPTER 6

Developer Guide

6.1 Project Structure

Each Django project consists of two things: a Django project, and one or more Django apps inside that project. For the What's Fresh API, there is only one app.

The project is named `whats_fresh`, and the app `whats_fresh_api`.

The Git repository contains the Django project as a subdirectory, with related files – the Vagrant setup file, the pip requirements file, etc in the root of the repository as well.

Inside the project folder, `whats_fresh`, there is a `setup.py` file that can be used to install the project. To manage the server, use `django-admin`. To read more about it and its functions, see the Django documentation.

Django stores the project information, including `settings.py`, inside the second project folder, `/path/to/repository/whats_fresh/whats_fresh`. The app itself is stored in `/path/to/repository/whats_fresh/whats_fresh_api`.

6.2 Issue Tracking

The bug tracker for the What's Fresh API is at code.osuosl.org, and all bugs and feature requests for the What's Fresh API should be tracked there. Please create an issue for any code, documentation or translation you wish to contribute.

6.3 Repository Layout

We loosely follow Git-flow for managing repository. Read about the branching model and why you may wish to use it too.

master Releases only, this is the main public branch.

release/<version> A release branch, the current release branch is tagged and merged into master.

develop Mostly stable development branch. Small changes only. It is acceptable that this branch have bugs, but should remain mostly stable.

feature/<issue number> New features, these will be merged into develop when complete.

When working on new code, be sure to create a new branch from the appropriate place:

- **develop** - if this is a new feature
- **release/<version>** - if this is a bug fix on an existing release

6.4 Code Standards

We follow PEP 8, "the guide for python style".

6.4.1 Developing with Docker

6.5 Platform dependent specifics

If you are using Linux you will need to prefix all of the following commands with sudo. If you are using OS X you will need to use the boot2docker tool.

6.6 Postgis image

The What's Fresh Docker workflow relies on the kartoza/postgis image available on the docker hub. To pull this image run:

```
$ docker pull kartoza/postgis
```

The image can take two optional environment variables to specify a user and password to the database. These will be specified with the -e option. A port should be provided with the -p followed by the port to communicate with the host machine, a colon, and the port to communicate with the container. Make sure the environment variables passed to this container match those which are passed to the What's Fresh API Docker image. Reasonable defaults can be found in the Dockerfile. Postgres typically runs on port 5432. To run the image:

```
$ docker run -d --name postgis -p $HOSTPORT:$CONTAINERPORT -e  USERNAME=$USERNAME -e PASS=$PASSWORD
```

Make sure that the What's Fresh project container connects to the database over the host port.

6.7 Building the What's Fresh docker image

```
$ docker build -t="osuosl/whats_fresh:dev" .
```

6.8 Running the What's Fresh docker image

The Dockerfile included in the root of the repository will load the code from the current directory. This means that any changes you made to your copy of the repository will be run. Environment variables can be passed with the -e option. The Dockerfile specifies a reasonable default set of environment variables, which can be overridden with the -e option.

Before the app is ready, create the database and run migrations.

```
$ docker exec -it postgis bash
# createdb -U $USERNAME -h localhost $DBNAME
# psql -U $USERNAME -h localhost
DBNAME=# create extension postgis;
CREATE EXTENSION
DBNAME=# ^D
# ^D
$ docker run --link postgis:postgis osuosl/whats_fresh:dev python manage.py migrate
```

Next, connect to the database with psql and create the relevant user.

```
$ psql -h localhost -U docker -p $HOSTPORT
```

Running the server is similar:

```
$ docker run --link postgis:postgis -p 8000:8000 osuosl/whats_fresh:dev
```

If you are running linux, connect to http://localhost:8000 in your browser. If you are running OS X, get the IP address of your boot2docker vm

```
$ boot2docker ip
192.168.59.103
```

Next connect to http://192.168.59.103:8000 in your browser.

On occasion it may be necessary to obtain a shell in the container:

```
$ docker run -it osuosl/whats_fresh:dev bash
```

Some developers may prefer to mount their copy of the application as a volume when they run the app:

```
$ docker run -v /path/to/code/:/opt/whats_fresh --link postgis:postgis osuosl/whats_fresh:dev
```

6.8.1 Developing

6.9 Requirements

This project comes with a Test Kitchen configuration set up to manage and create a homogeneous development environment and allow developers to destroy and recreate their environment in the case that something goes horribly, horribly wrong. It's not necessary to use this environment, but using it will make sure that your environment is as close to the production environment, and to other developer's environments, as possible.

To set up a development environment yourself, see *Manually setting up the What's Fresh environment*.

To set up this environment on your own machine, you'll need a few things:

Chef DK

The first step of this process is to install the Chef Development Kit. It can be obtained from getchef.com

Ruby Gems

In order to install the required gems, you'll need to install the ruby

Kitchen is a Ruby gem. To install it, just use `gem install`:

```
$ chef gem install knife-spork knife-flip knife-solve knife-backup knife-cleanup \
  knife-env-diff foodcritic berkshelf test-kitchen kitchen-vagrant kitchen-openstack
```

Vagrant

To install Vagrant, just use your package manager:

```
$ sudo yum install vagrant # Debian or Ubuntu
$ sudo apt-get install vagrant # Centos
```

vagrant-berkshelf and vagrant-omnibus

These plugins are used to configure the Vagrant machine. To install these plugins, you'll need to use Vagrant's plugin manager:

```
$ vagrant plugin install vagrant-berkshelf
```

Berks

Now, you'll need to update your Berkshelf. This allows your virtual machine to configure itself:

```
$ berks update
```

You're ready to go! To get the environment started, type `kitchen converge dev` in the root of the Git repository.

After a while (this process may take a quite few minutes), your machine will be ready to use. To log in, type `kitchen login dev`.

Now you should be on the Vagrant machine:

```
[vagrant@develop-centos-65 ~]$
```

To get developing, you'll need to prepare your virtual environment. To do so, first activate the Python virtualenv:

```
[vagrant@develop-centos-65 ~]$ source /opt/whats_fresh/shared/env/bin/activate
```

Your prompt should look like this now:

```
(env)[vagrant@develop-centos-65 ~]$
```

6.10 Manually setting up the What's Fresh environment

The What's Fresh API has been developed and tested on Python 2.7, Postgres 9.3.5, and PostGIS 2.1.3, with GDAL 1.9.2.

Installing PostGIS and requirements

To install PostGIS, PostgreSQL, and its requirements, follow the installation instructions on PostGIS's website.

After installing PostGIS and Postgres, you'll need to prepare the database using the `psql` tool:

```
$ createdb whats_fresh
$ psql whats_fresh
whats_fresh-# CREATE EXTENSION postgis;
```

You can exit the PSQL prompt by pressing Ctrl+D on your keyboard.

Getting What's Fresh source code

After PostGIS is installed, you'll need to use `git` to clone the What's Fresh repository. If you don't have `git`, install it using your system's package manager.

Now, clone the API repository:

```
$ git clone https://github.com/osu-cass/whats-fresh-api.git
```

This will place the source code in the subdirectory `whats-fresh-api`. You'll want to use a Python virtual environment and the `pip` package manager to set up the Python requirements:

```
$ cd whats-fresh-api
$ virtualenv ~/.virtualenvs/whats-fresh
$ source ~/.virtualenvs/whats-fresh/bin/activate
(whats-fresh)$ pip install -r requirements.txt
$ cd whats_fresh
```

You're now ready to run and develop the project!

6.11 Running the Django project

At this point, you should have a working database and copy of the source code. You may be developing on your physical machine, or using a virtual machine as described above. After setting up the virtual environment, navigate to the project directory, and install the server using `setup.py develop`:

```
(env)[vagrant@develop-centos-65 ~]$ cd whats_fresh/
(env)[vagrant@develop-centos-65 whats_fresh]$ python setup.py develop
```

Now, you can run the `django-admin` tool from anywhere in your environment. However, you'll need to tell it what `django-settings` to use by exporting the proper environment variable:

```
(env)[vagrant@develop-centos-65 whats_fresh]$ export DJANGO_SETTINGS_MODULE="whats_fresh.settings"
```

Create the database tables using `django-admin`:

```
(env)[vagrant@develop-centos-65 ~]$ django-admin migrate
```

If you plan on logging into the web interface, you'll need to create a user account. You can use `django-admin` to create a superuser account:

```
(env)[vagrant@develop-centos-65 ~]$ django-admin createsuperuser
```

You should now be ready to run the Django app!

```
(env)[vagrant@develop-centos-65 ~]$ django-admin runserver 0.0.0.0:8000
```

To access the server in your web browser, navigate to `http://172.16.16.2:8000`.

6.12 Testing

The What's Fresh API uses test-driven development. What this means is that, before writing a feature – be it a new API endpoint, a model, or a bug fix – you should write a test. After writing the feature, run the test to verify that it works, and when you're satisfied with your implementation, re-run the entire test suite to make sure there were no regressions.

Each test lives inside the `whats_fresh_api/tests/` directory, organized into a subdirectory based on what kind of test it is. For instance, all model tests live inside the `models` subdirectory, while views would live inside the `view` directory.

For information on how to write tests, see Django's guide on writing tests.

Let's say you've just modified the code – say, you edited the Vendor model due to a bug you found. Instead of running the entire testing suite, you can run just one set of tests at a time:

```
(env)[vagrant@develop-centos-65 whats_fresh]$ django-admin test whats_fresh.whats_fresh_api.tests.mod
```

Note: Running tests is based on the directory name, using the following syntax:

```
whats_fresh.whats_fresh_api.tests.<test subdirectory>.<test file>.<test class name>
```

For a test called ImageTestCase inside of `tests/views/test_image_view.py`, you would need to run the following command:

```
(env)[vagrant@develop-centos-65 whats_fresh]$ django-admin test whats_fresh.whats_fresh_api.tests.vie
```

To make sure that you didn't break anything unexpected, it can be a good idea to periodically run the entire testing suite:

```
(env)[vagrant@develop-centos-65 whats_fresh]$ django-admin test whats_fresh
```

Fixtures

Django allows you to load pre-written data into the database for testing purposes. The data is stored in files called fixtures, and for testing purposes, the What's Fresh API comes with a few hand-written (for running tests where we need to know the input data) and a large number of automatically generated (for when we simply want to have data in our database).

To install a fixture, use the `django-admin` command's loaddata option:

```
(env)[vagrant@develop-centos-65 whats_fresh]$ django-admin loaddata fixtures
```

There are many sets of fixtures available. `test_fixtures` is the original set of fixtures, but the `real_data` fixtures are more comprehensive and should be used in new tests.

Model and View documentation:

CHAPTER 7

Models

The project holds the data in the database using the following models:

CHAPTER 8

Views

8.1 Endpoints

The API uses the following views for endpoints:

8.1.1 Helper Functions

In addition, there are some helper functions stored in `functions.py`:

exception `whats_fresh.whats_fresh_api.functions.`**`BadAddressException`**
The exception thrown if the address passed in invalid.

`whats_fresh.whats_fresh_api.functions.`**`coordinates_from_address`**(*street*, *city*, *state*, *zip*)
This function returns a list of the coordinates from the address passed using the Google Geocoding API. If the address given does not return an exact coordinates (for instance, if the address can only be located down to the city), a BadAddressException is thrown.

TODO: this should probably return a tuple, rather than a list.

`whats_fresh.whats_fresh_api.functions.`**`get_lat_long_prox`**(*request*, *error=None*)
Parse the latitude, longitude, proximity, and limit for the Vendor list functions.

If the parsing results in an error, the error block is updated to reflect that error.

`whats_fresh.whats_fresh_api.functions.`**`get_limit`**(*request*, *error=None*)
Return the limit requested by the user.

If the limit results in an error, the error block is updated to reflect that error.

`whats_fresh.whats_fresh_api.functions.`**`group_required`**(**group_names*)
This decorator can be used to protect a view from users not in a given list of groups. Add @group_required to a view to require the user to be logged in and part of the passed groups. If the user is not a member of the given groups, they will be redirected to /login.

8.2 Data Entry

The backend, data-entry interface uses the following:

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*

W

B

C

G

W