Modeler: Web API Modeling Tool

A Senior Project

presented to

the Faculty of the Computer Science Department

California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science in Computer Science

by

Yegor Pomortsev

March, 2014

Contents

1 Introduction

1.1 Problem

Modeler aims to solve the problem of creating web API models more quickly and easily than possible with existing tools and technologies. A secondary goal of the project is to explore a solution for creating API models semi-automatically from a sample of traffic data.

1.1.1 Background

Web APIs provide a way for web applications or services to provide third-party software (clients) with access to their data or operations on the data, through a protocol typically built on top of standard web protocols like HyperText Transfer Protocol (HTTP) [1] and HTTP Secure (HTTPS) [2] and data formats like JavaScript Object Notation (JSON) [3]. Communication over this protocol can often be described abstractly as a collection of requests from the client to the API and responses back. Each request is describes a method to be performed by combining a resources (piece of data), and a verb, which describes the operation to be performed on that data. For example, a method to retrieve a list of to-do items from an example service consists of a verb, GET, and a resource, `/todos` (see appendix B for a description of this API).

An API model is a formally describes the API's protocol, including its functions (methods), data formats, authentication schemes, and any other metadata, such as human-readable documentation or sample requests and responses. For web APIs that follow the resource/method abstraction, the model describes each resource and related methods. The model itself can be described using an API modeling language/format, such as Web Application Description Language (WADL) [4] or Web Services Description Language (WSDL) [5] – *Modeler* uses the API Modeling Service's (see appendix A) JSON-based model format.

1.2 Solution

The *Modeler* tool is a web application that allows users to view, create, modify, and analyze web API models through a graphical user interface (GUI), alleviating the need to edit a model using its usual text-based format. Additional system features, like automatically creating a model from a sample of web API traffic reduce the amount of work that a user has to do to complete several typical use-cases related to web API modeling (see section 2).

The major goals of this project are to:

- Provide a web application interface to the API Modeling Service API (see appendix A) that allows users to interact with API models stored using this service.
- Extend the API Modeling Service API with expert system functionality for analyzing, creating, and augmenting API models using raw request/response traffic.

Fundamentally, the *Modeler* system is a combination of a backend, which extends the API Modeling Service API with extra functionality, and a web application frontend (see section 3 for an in-depth description of the system design and implementation).

1.3 Document Conventions

HTTP paths and Uniform Resource Locators (URLs) are presented in a monospaced font:

```
https://api.example.com/v1
```

Pre-formatted data and program source code is presented as a block of monospaced type:

```
This is a block of code or other formatted
data. The ↪ symbol at the start of a
line denotes wrapping from the
previous line.
```

See the Glossary and Acronyms sections at the end of the document for descriptions of technical terms and acronyms used throughout this document.

2 System Use-Cases and Usage Scenarios

This section presents several common use-cases and usage scenarios for web API models, and their solutions both with and without using the *Modeler* tool:

- creating a new model from scratch
- generating human-readable API documentation from an annotated model
- generating an API client library that allows structured, programmatic access to the API, and:
- creating an API model from an existing API traffic (sampled from communication between a client and the API)

2.1 Creating an API Model

The *Modeler* tool's core functionality is to provide an interface for creating new API models and editing existing ones. The following sections compare how to create a model and store it in the back-end service with and without using the *Modeler* frontend.

2.1.1 Without *Modeler* (command line)

To create an API model and save it to the API Modeling Service backend, the user first needs to write the model in the JSON format accepted by the backend, using a standard text editor or integrated development environment (IDE). See appendix B for an example of a model in the JSON format.

After creating the model, the user then needs to save it to the API Modeling Service. This can be done by manually issuing calls to the API Modeling Service API using the Curl [6] command-line tool. First, the user registers the API:

```
curl -X POST \
     -H Content-Type:application/json \
     -H Authorization:Basic ... \
-d '{ "name": "todos", \
      "displayName": "To-Dos", \
      "description": "Keeps track of all the things you need to do" }' \
https://modeler/v1/o/test/apimodels
```

In this example, the API Modeling Service API is available at the URL `http://modeler/v1`, and the authorized user belongs to the `test` organization.

The user then uploads the API model, saved as the file `todosapi.json`, by creating a new revision of the API:

```
curl -X POST
     -H Content-Type:application/json
     -H Authorization:Basic ... \
     -d '@todosapi.json'
https://modeler/v1/o/test/apimodels/revisions
```

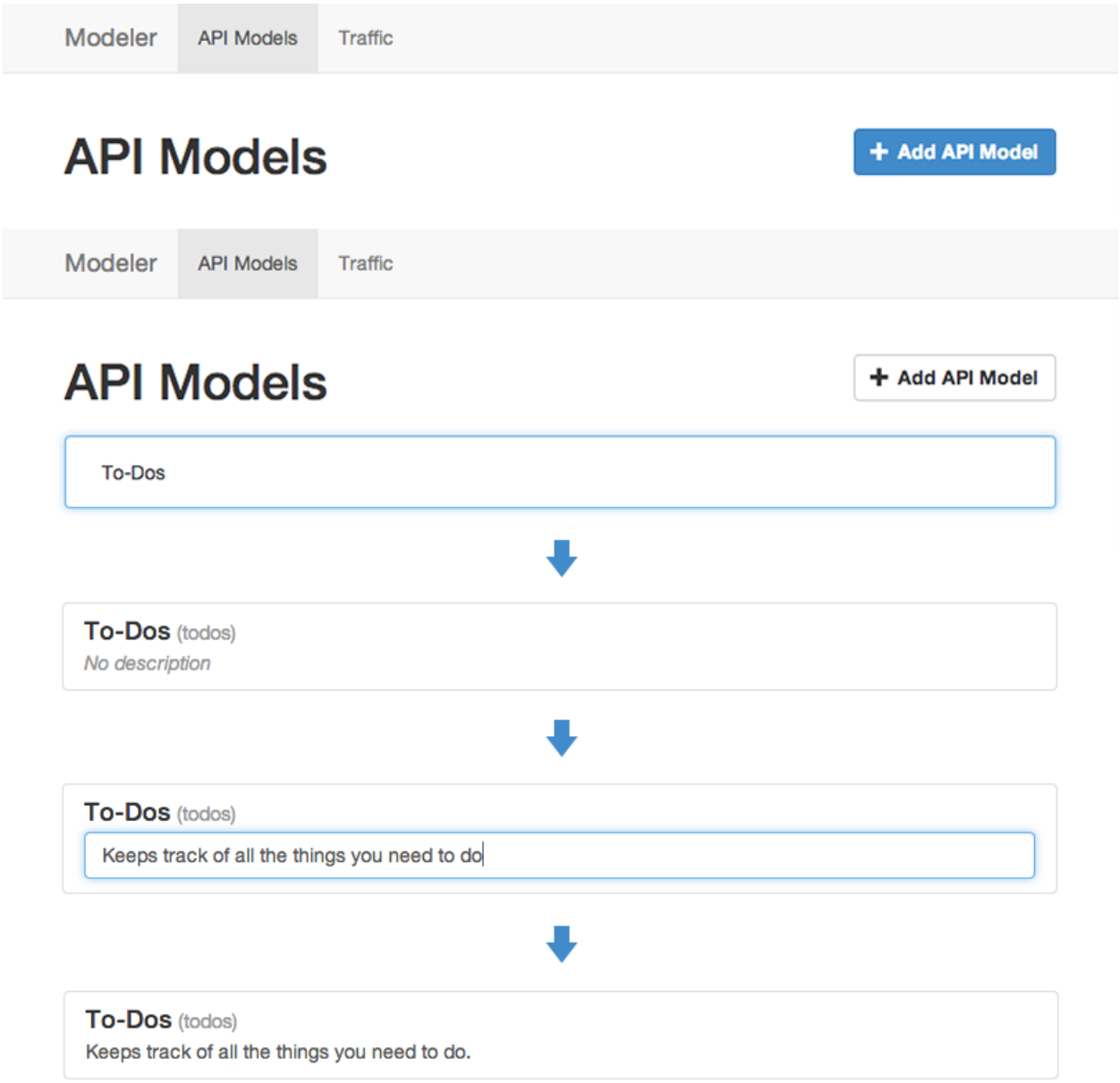

Figure 1: Creating a new API model

2.1.2 With *Modeler*

Creating a new API model is done using the *Modeler* web frontend interface. Figure 1 shows the process of creating a new, empty API Model – the user starts by clicking the "Add API Model," then enters a name and hits Enter, which creates the model. The user can then edit the description by clicking the "No description" text, entering a new description, and hitting Enter.

The user then selects the new API model from the list and creates a new revision of the API model. Each revision is itself a separate model and its contents don't depend on any other revisions of that API. Figure 2 shows the process of creating a new revision – the user clicks the

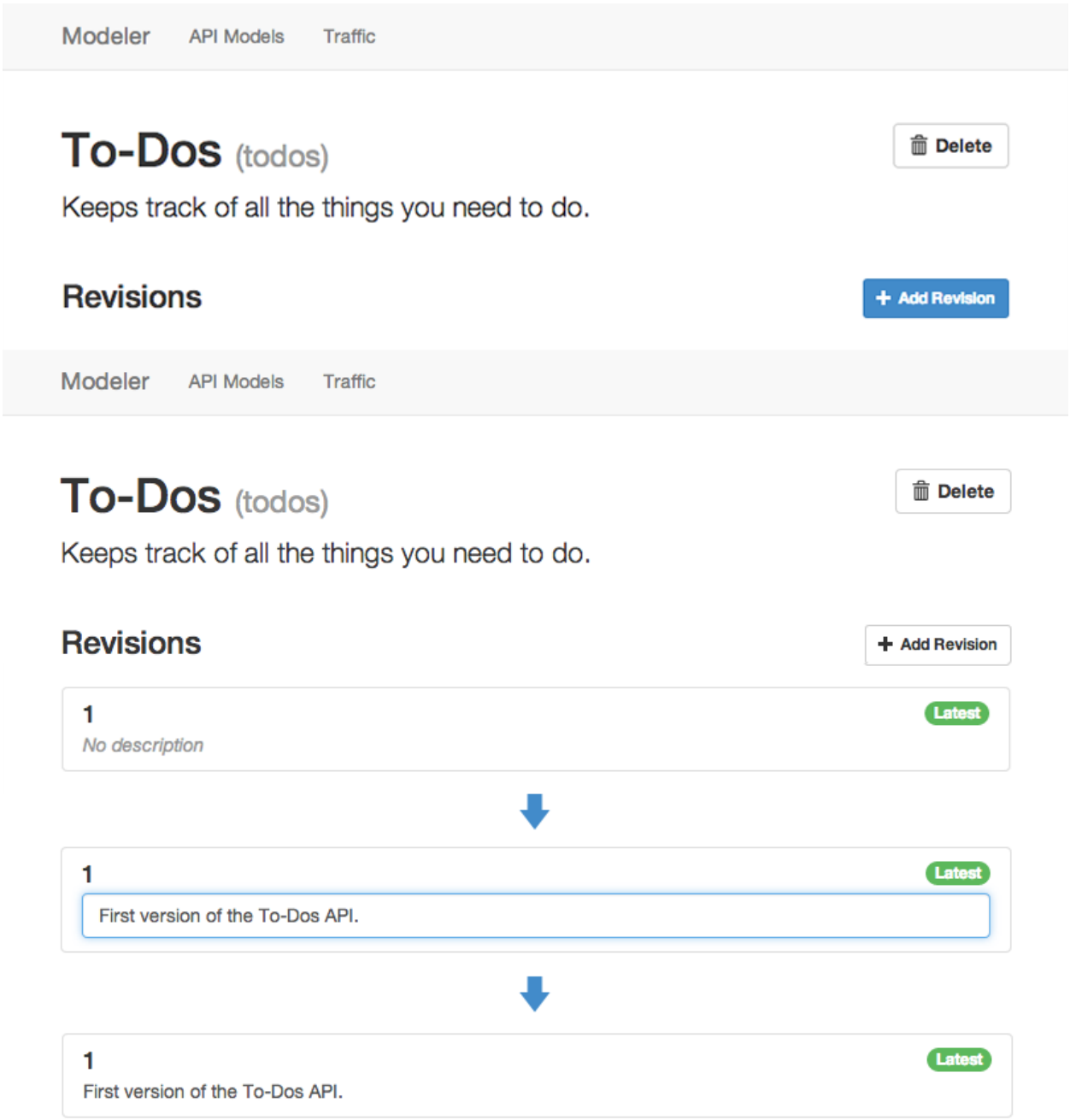

Figure 2: Creating a new API revision

"Add Revision" button, which creates a new empty revision. The user can then change the revision's description as they did in figure 1 when creating the API model.

Afterwards, the user selects the newly-created revision and edits it, adding parameters, resources, methods, and other API model properties. Figure 3 shows the revision editing interface populated with details of the example API. The editing process is similar to creating a new API model or revision – clicking the "Add Parameter," "Add Resource," and "Add Method" prompt the

user for a name of the new parameter, resource, or method being created. After the user enters a name, an empty, named element is created, which can then be edited by clicking the appropriate fields and buttons within the element.

2.2 Generating API Documentation

API documentation is a document or set of documents that describes how the API should be accessed and used in third-party software, but not necessarily how the functions that it provides are implemented. At the most basic level, an API model is itself a form of documentation for an API– it gives the technical details of all the functions that the API provides and the input and output formats that it uses. However, a raw model in its native machine-readable format is not an ideal form of API documentation that is meant to be consumed by (human) software developers. Instead, the model can be converted into a human-readable document that is easier to read. Additionally, the documentation can provide content and features that help the developer understand the API like sample requests and responses, or a sandbox environment to easily test API functionality.

This functionality is similar to documentation generator software like Doxygen [7], which uses source code annotated with comments, examples, and usage instructions as input.

The documentation for a particular API model revision is available by accessing the appropriate (URL in a web browser. This functionality is provided by the API Modeling Service. For example, to show documentation for revision 1 of the `todos` API model that resides in the `test` organization, the URL is: `https://modeler/v1/o/test/apimodels/todos/revisions/1/doc`

Figure 4 shows the index page of the generated documentation for the example API.

Figure 5 shows a detail page for one of the resources/methods described by that model. The detail page provides a tool to test API requests directly from the documentation – the user sets values for any parameters (e.g. `todoId`), and clicks the "Send this request" button, which calls the API and displays the result.

2.3 Generating an API Client Library

API client libraries are used to simplify creating requests to an API from a piece of software by providing a bridge between the protocols and formats used by the API (e.g. HTTP for data transfer and JSON as the data format) and the facilities native to the programming language the client software is written in. The client library can provide functions to convert and format data to the format expected by the API, enforce restrictions, and reduce the overall amount of code that needs to be written.

Because web APIs generally provide their services through HTTP, they can be accessed by making HTTP requests directly. For example, a program written in the Python [8] programming language can access the example API using the built-in `urllib2` HTTP library:

```
import urllib2
response = urllib2.urlopen(
    'http://api.example.com/v1/todos/1')
todo = response.read()
```

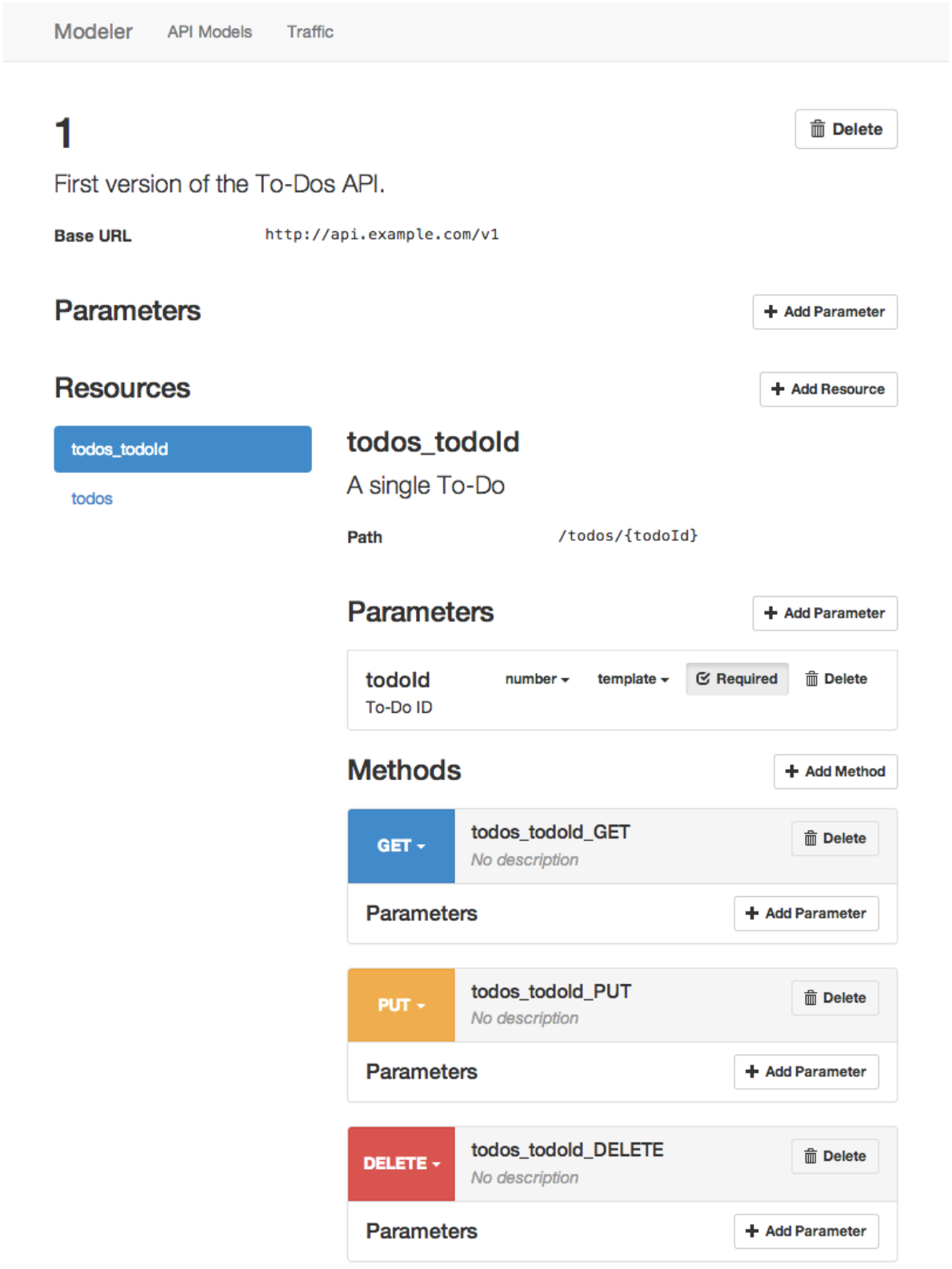

Figure 3: Editing an API revision

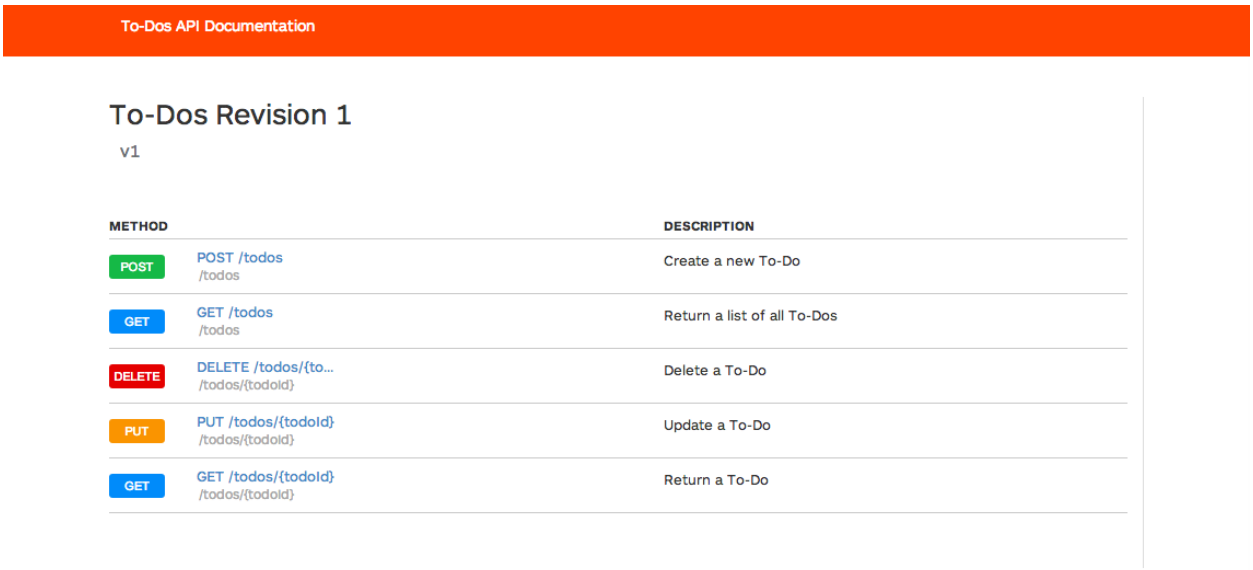

Figure 4: Generated API documentation index page

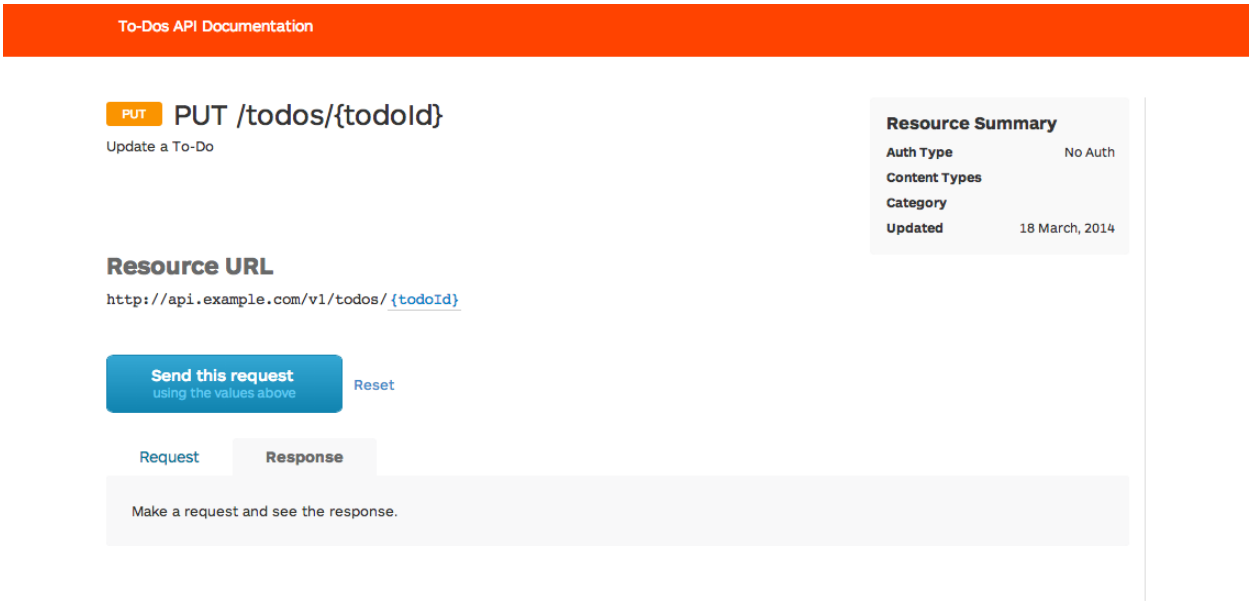

Figure 5: Generated API documentation method detail

However, the resulting data will be returned in the JSON format, which requires further processing to read. Similarly, to create a new to-do, the client would have to format the request in JSON. This can be simplified with a (hypothetical) library interface to our example API:

```
import todosapi
api = todosapi.Client(
    'http://api.example.com/v1')
todo = api.todos[1]
```

The `todosapi` library handles all the necessary communication and API resource path construction, providing the programmer with an easy-to-use interface to the API.

To generate an API client library, the user would select an API model revision and the language/platform the desired library should be created for within the *Modeler* GUI, and trigger the generator feature. The system would then return a generated client library as a downloadable source code archive.

The current version of *Modeler* does not implement an API client library generator – this functionality can be part of future work on the system.

2.4 Creating an API Model from API Traffic

Some properties of an API can be inferred from analyzing a sample of API traffic in order to create a model of that API from scratch or modify an existing model. For example, given the following sample traffic, represented as a chronological list of (simplified) HTTP requests, consisting of a verb and an URL:

```
GET http://api.example.com/v1
GET http://api.example.com/v1/todos
POST http://api.example.com/v1/todos
GET http://api.example.com/v1/todos/1
GET http://api.example.com/v1/todos/2
PUT http://api.example.com/v1/todos/1
```

The *Modeler* system can infer that the API has the base URL `http://api.example.com/v1`, since every request made is rooted at that URL. Additionally, the traffic shows that there exists a `/todos` resource that has methods with the GET and POST verbs.

If the response content for each of the requests above is available, the system can make further guesses about the structure of the API. Specifically, a common pattern for Representational State Transfer (REST) APIs that *Modeler* targets is to access resources inside collections through a subpath in the URL – in the example API, resources (todo objects) inside the collection `/todos` can be referenced by appending their identifiers (IDs) to the URL: e.g. `/todos/1` and `/todos/2`. If the response to the resource `/todos` returns a "list-like" or "collection-like" object (e.g. a JSON array), the system may assume that it is a collection, and that its children resources (e.g. `/todos/1`) are elements of that collection. This way, the two unique resources above, `/todos/1` and `/todos/2` can be consolidated into a single, parameterized resource: `/todos/{todoId}`, where `{todoId}` is a template parameter that represents the ID of a to-do object in the collection. Similarly, the two separate methods for the resource `/todos/1` represented by the requests using the GET and PUT verbs can be modeled by the methods `GET /todos/{todoId}` and `PUT /todos/{todoId}`.

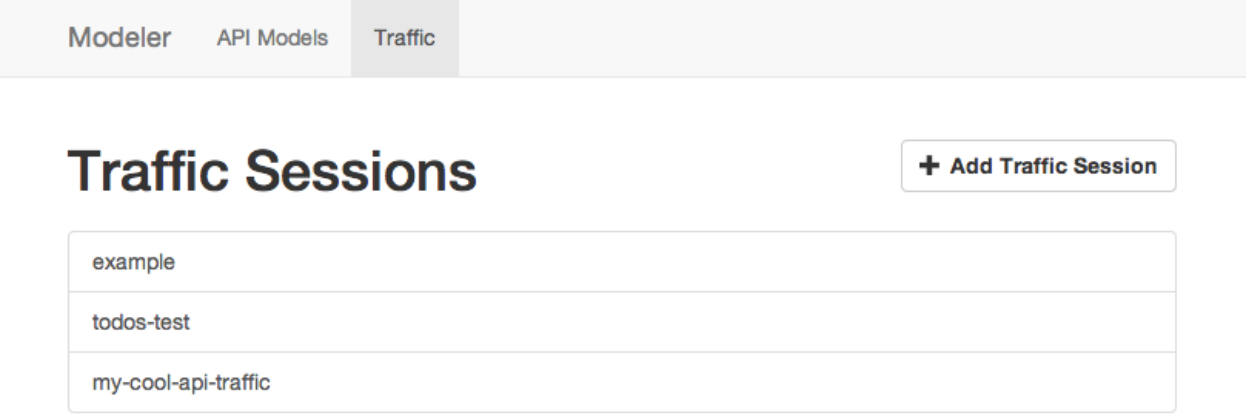

Figure 6: List of traffic sessions (before adding new session)

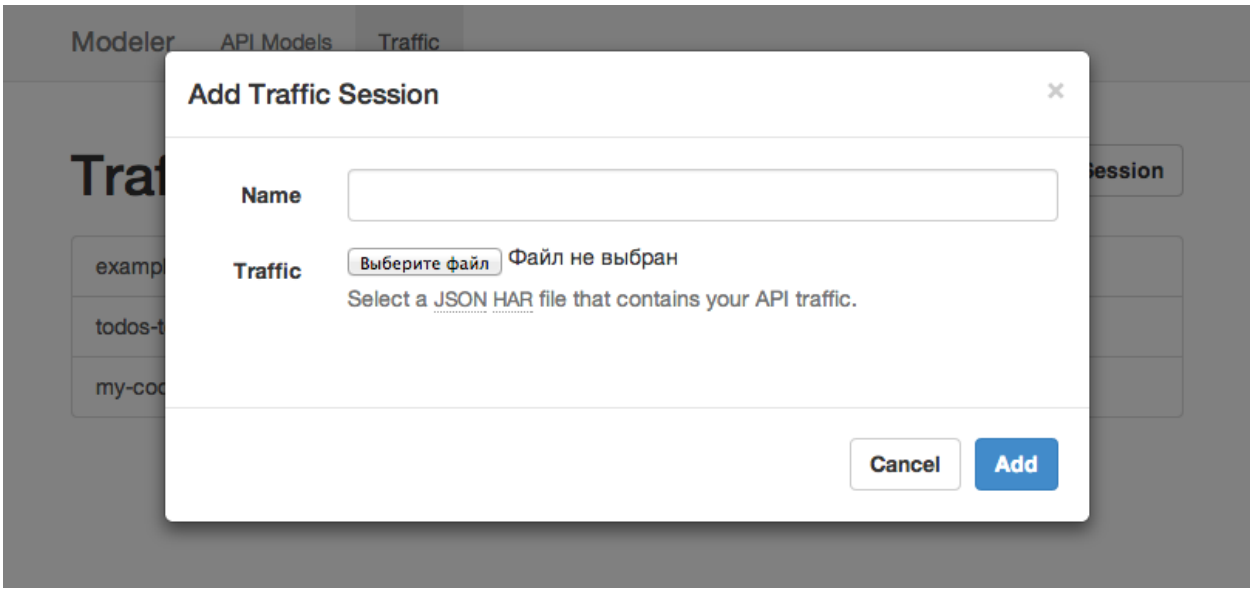

Figure 7: Adding a new traffic session (before user input)

In *Modeler*, API traffic is stored as a "session," which can be associated with a particular API revision. Users can upload new traffic sessions and create API models from one or more traffic session, or associate the sessions with an existing model to provide suggestions for changes to the model. Currently, only the functionality to upload new traffic sessions is implemented – future versions of *Modeler* will use the traffic sessions for analysis.

Figure 6 shows the *Modeler* GUI screen that lists traffic sessions stored in the system. The user can add a new traffic session by clicking the "Add Traffic Session" button.

Figure 7 shows the dialog window in which the user enters details of the new traffic session.

Figure 8 shows the dialog after the user has entered a name for the new session and selected a traffic file in HTTP Archive (HAR) format. The traffic session is created when the user clicks the "Add" button.

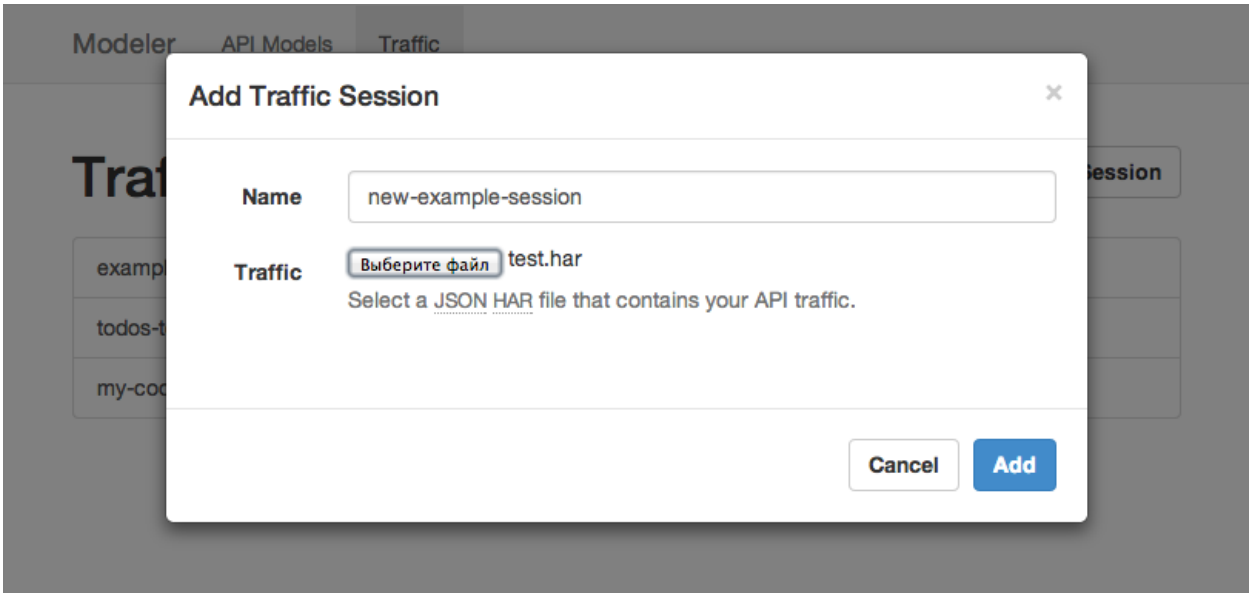

Figure 8: Adding a new traffic session (after user input)

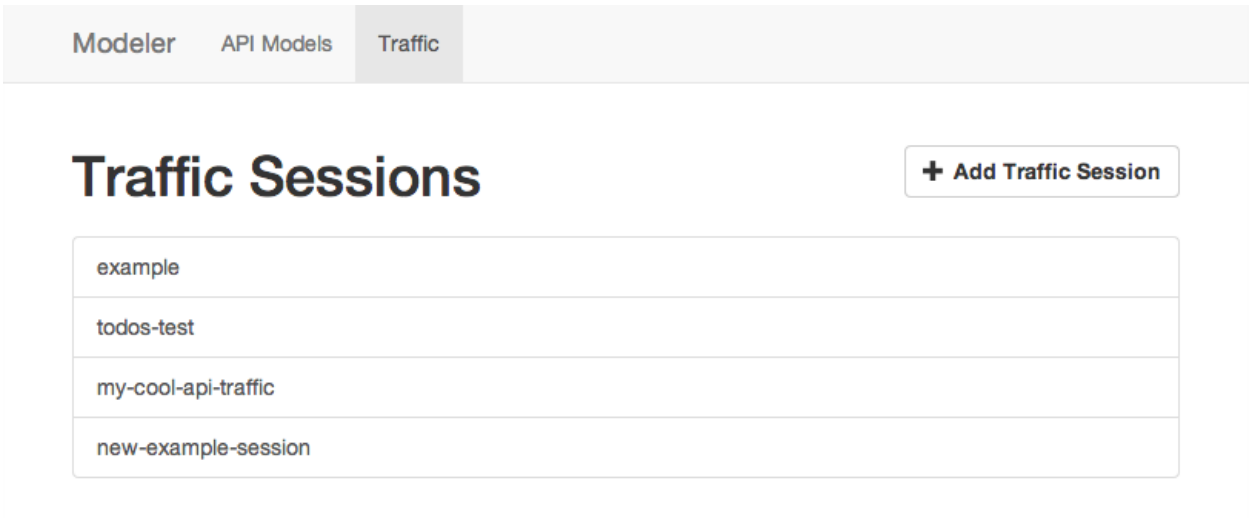

Figure 9: List of traffic sessions (after adding new session)

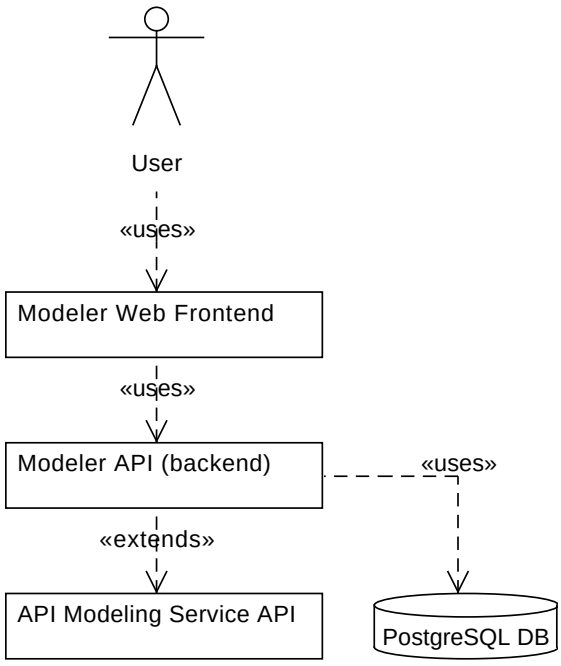

Figure 10: *Modeler* software architecture overview

Figure 9 shows the list of traffic sessions with the newly created traffic session.

3 Design and Implementation

At the highest level, *Modeler* consists of two major subsystems/components: an API backend, and an interactive web application frontend.

Figure 10 shows an overview of the system architecture: users interact with the *Modeler* system through the web frontend, which uses the API backend. The API backend extends the API Modeling Service (see appendix A) and uses a PostgreSQL database to store data that is not handled by the API Modeling Service (e.g. traffic sessions).

3.1 Web Frontend

The *Modeler* web frontend is a web application that is accessed through the Internet using a typical web browser. The frontend is developed using standard web technologies: HTML, CSS, and JavaScript. Specifically, the AngularJS [9] framework is used to structure the application in conjunction with the Sass [10] stylesheet extension language and the Bootstrap [11] user interface (UI) framework.

Figure 11 shows an overview of the *Modeler* frontend software architecture. The application is structured using the AngularJS concepts of controllers, views, and directives, which model a version of the classic Model-View-Controller (MVC) architecture [12].

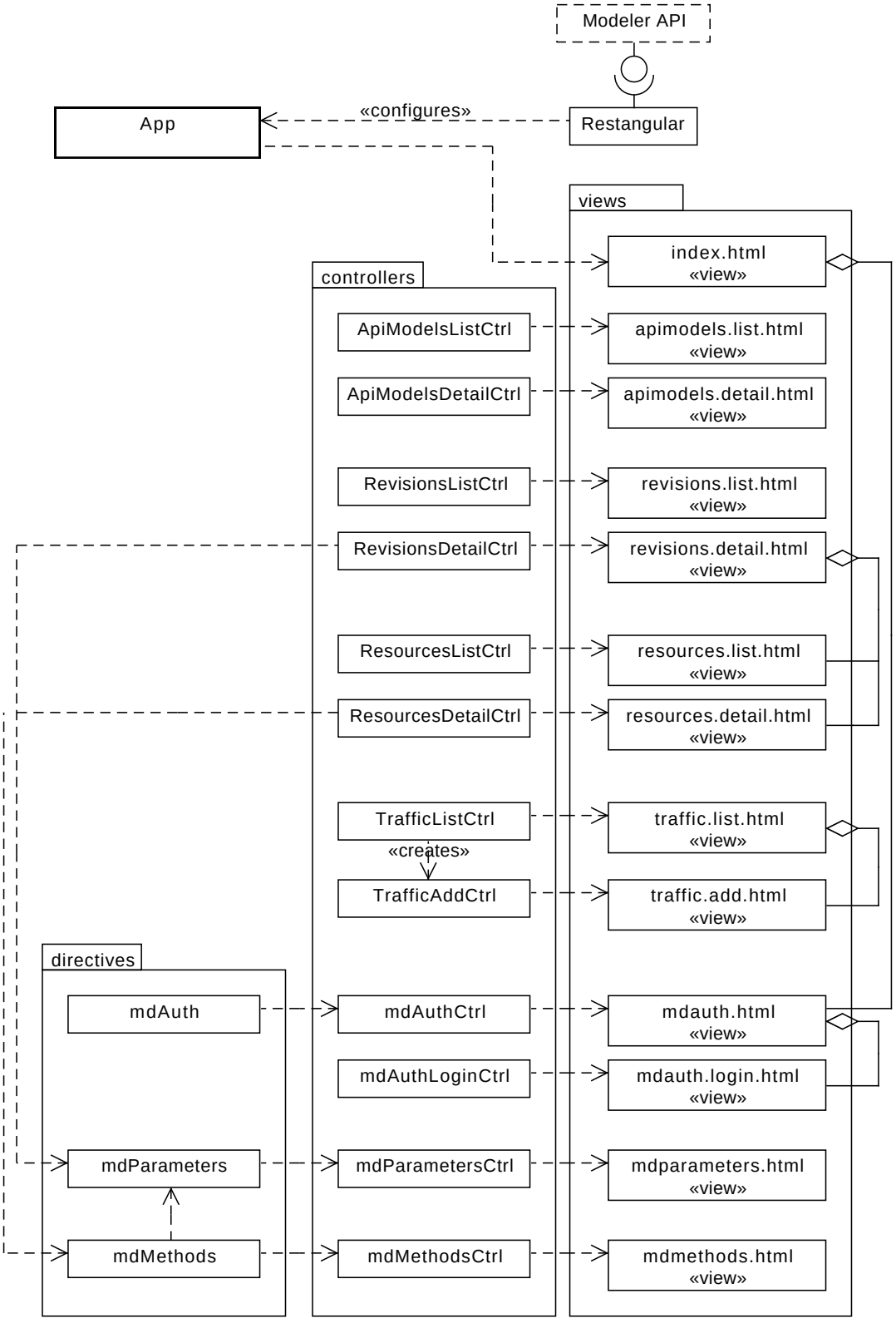

Figure 11: *Modeler* frontend software architecture overview

3.1.1 Implementation Details

The following code snippet shows a portion of the application configuration performed in the App module seen in figure 11:

```
$stateProvider
  .state('apimodels', {
    abstract: true,
    url: '/apimodels',
    template: '<ui-view/>'
  })
  .state('apimodels.list', {
    url: '',
    templateUrl: 'views/apimodels.list.html',
    controller: 'ApiModelsListCtrl'
  })
  .state('apimodels.detail', {
    url: '/{modelName}',
    views: {
      '': {
        templateUrl: 'views/apimodels.detail.html',
        controller: 'ApiModelsDetailCtrl',
      },
      'revisions.list@apimodels.detail': {
        templateUrl: 'views/revisions.list.html',
        controller: 'RevisionsListCtrl'
      }
    }
  });
```

The `$stateProvider` variable is an instance of the AngularUI Router [13] framework, which manages application state, instantiates controllers and views, and handles URL routing. In the context of the *Modeler* frontend, states are generally separate screens or windows in the application – in the example above, the `apimodels` state contains two sub-states: one to list API Models (i.e. the screen shown in figure 1, and another to display details about and edit a single API model (including creating a new revision, as shown in figure 2).

Each state definition specifies the controller and `templateUrl` (view) to be used for that state. States can contain several sub-views, like the `apimodels.detail` state, which contains a list of API revisions that has a separate controller and view.

The following code snippet shows the implementation of the `ResourcesList` controller, which works in conjunction with its corresponding view and the *Modeler* API backend to show a list of resources in an API revision:

```
angular.module('modelerApp')
.controller('ResourcesListCtrl', function($scope, Restangular, $stateParams) {
   $scope.resources = { _:
    Restangular
      .one('apimodels', $stateParams.modelName)
      .one('revisions', $stateParams.revisionNumber)
```

```
      .all('resources')
      .getList()
      .$object
  };
});
```

The `ResourcesList` depends on the current scope, which is used to communicate data to the view, the `Restangular` module, which is used to communicate with the backend API, and the current state parameters (i.e. which API model and revision the user has selected). The controller retrieves a list of resources for the particular API model and revision for the current state, and creates a scope variable that contains this list.

The following HTML snippet shows an abbreviated version of the resources list view used in conjunction with the controller described above:

```
<ul>
  <li ng-repeat="resource in resources._"
      ui-sref-active="active">
    <a ui-sref="apimodels.detail.revisions.detail.resources.detail({ resourceName:
       ↪ resource.name })">
      {{ resource.name }}
    </a>
  </li>
</ul>
```

The above snippet is an HTML template used to generate the GUI that shows a list of API revisions. For each revision in the list of revisions, it creates an HTML list item with a single link containing the name of the resource. The additional `ui-sref-active` and `ui-sref` attributes are used to generate a visual indicator of the currently-selected resource, and a clickable link to transition to the `resources.detail` state, which displays the selected revision.

3.2 API Backend

The *Modeler* API backend is a server application developed the Node.js [14] platform using the JavaScript programming language.

Figure 12 shows an overview of the *Modeler* API software architecture. The system consists of a central `API` object that performs request handing, routing, and configuration of child classes. Separate classes provide response handing for the various resources that are handled by the *Modeler* API – unresolved requests are forwarded to the backend API Modeling Service API (see appendix A). The system maintains a connection to a separate PostgreSQL database to provide storage for data not handled by the API Modeling Service (e.g. API traffic sessions, which are manipulated through the `SessionsResource`).

3.2.1 Implementation Details

The following code snippet shows the initialization section of the `SessionsResource` module that provides Create-Read-Update-Delete (CRUD) functionality for API traffic sessions:

```
module.exports = function(Session, Sessions, config) {
```

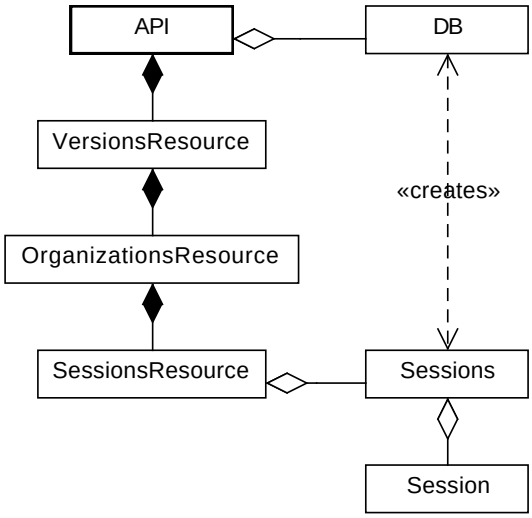

Figure 12: *Modeler* API software architecture overview

```
  var SessionsResource = function() {};

  SessionsResource.prototype.init = function(resourceConfig) {
    resourceConfig
      .produces('application/json')
      .consumes('application/json')
      .path('/sessions')
      .get('/', this.list)
      .post('/', this.create)
      .get('/{session}', this.show)
      .put('/{session}', this.update)
      .patch('/{session}', this.patch)
      .del('/{session}', this.remove)
  };

  // ...
};
```

The module depends on the `Session` and `Sessions` modules that are passed to the constructor in a dependency injection style. The module then declares the `SessionsResource` object and its initializer, `init`. The initialization function receives the `resourceConfig` object that is used to describe the HTTP interface that this module uses. In particular, the `init` section states that this resource produces and consumes data in JSON format, and has the HTTP path `/sessions`. The following HTTP verb and parameterized path statements declare which methods are called when the client makes a request to that path.

The following code snippet is the implementation of the `update` function, which corresponds to the HTTP PUT verb and updates an existing traffic session:

```
/** Update a session */
SessionsResource.prototype.update = function(env, next) {
env.target.skip = true;

Promise.all([
    jsonBodyAsync(env.request),
    Sessions.getByIdOrName(env.route.params.session)
  ])
  .spread(function(body, session) {
    if (typeof body.id !== 'undefined') {
      delete body.id;
    }

    return session.save(body);
  })
  .then(function(session) {
    env.response.body = session.toJSON();
    next(env);
  });
};
```

The `update` function receives the current environment, `env`, which contains information about the current HTTP request and response, and a `next` function that it calls when it has finished processing the request. The first line prevents this request from being forwarded to the underlying API Modeling Service API. The function then converts the request body content (the new traffic session) from JSON, and fetches the traffic session by the ID or name given in the request URL. Afterwards, it saves the session with the new data (removing any ID attribute to prevent the ID from being modified), and returns the new traffic session back to the client in the response body, in JSON format.

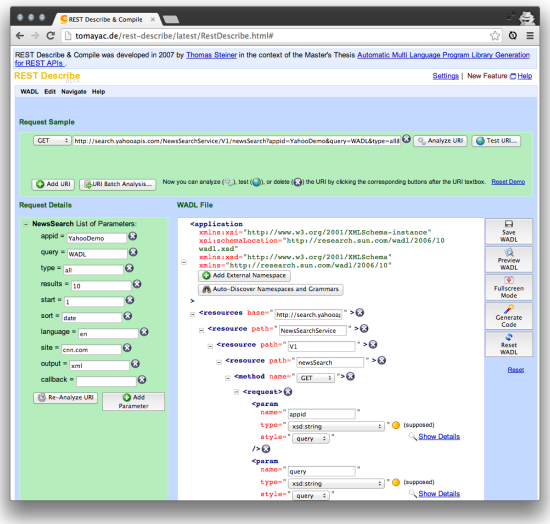

Figure 13: REST Describe & Compile GUI

4 Related Work

4.1 REST Describe & Compile

An inspiration for *Modeler* is the REST Describe & Compile [15] web application created by Thomas Steiner as part of his thesis on automatic program library generation for REST APIs [16], which allows creation of API models using analyzed API traffic and editing of the generated model, described in the WADL [4] format.

This is similar to the proposed *Modeler* API model generation from API traffic feature (see section 2.4). Like REST Describe & Compile, the goal of the *Modeler* functionality is to automatically determine the resources, methods, and parameters in an API using a sample of raw traffic. However, *Modeler* also provides a graphical editing interface for the resulting model – REST Describe & Compile allows editing, but with a limited interface that is strongly tied to the underlying WADL model format. *Modeler* does not yet have the proposed API client generation capabilities (see section 2.3) that REST Describe & Compile has.

Figure 13 shows the REST Describe & Compile GUI.

4.2 Swagger

Swagger [17] is an API model description language and toolset for generating API documentation. Figure 14 shows the Swagger API documentation interface, which contains a built-in client for making API test requests.

Like *Modeler* and the API Modeling Service, Swagger uses a JSON-based API model format, and has a documentation generator.

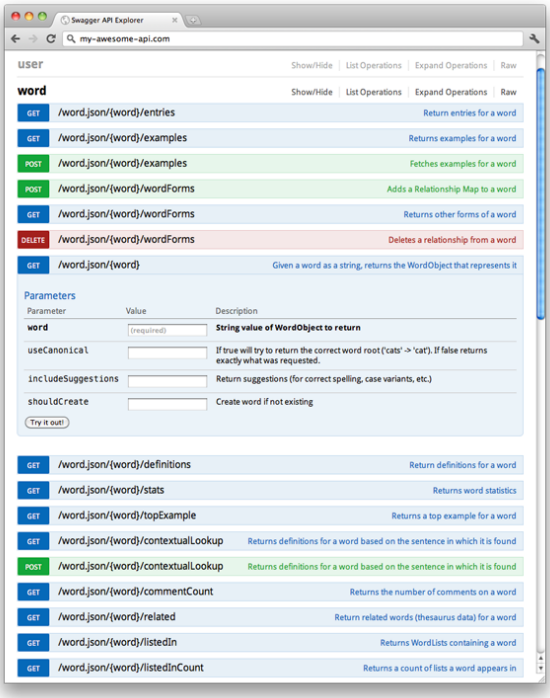

Figure 14: Swagger API Documentation GUI

5 Future Work

Future work to the *Modeler* system can be considered in the following areas:

- API model editing
- Generating API client libraries (section 2.3)
- Creating API models from API traffic (section 2.4)
- Exposing API Modeling Service functionality in frontend GUI

5.1 API Model Editing

Currently, the *Modeler* frontend displays most of the API model data returned by the backend, and allows editing a subset of it. Particularly, it provides basic CRUD functionality for API models, revisions, resources, methods, parameters (in revisions, resources, and methods), and traffic sessions.

The following model data is currently not being displayed, and cannot be modified through the *Modeler* frontend: 1) creation/modification timestamps 2) tags 3) custom attributes 4) parameter groups 5) API schemas 6) authentication schemes 7) method body/response descriptions 8) method samples 9) API groups 10) custom templates

5.2 Generating API Client Libraries

This feature, as described in section 2.3, is currently not implemented.

Implementing it would involve extending the *Modeler* backend API with the appropriate resources and methods to trigger generating a client library from a particular API revision given a target programming language or environment, and exposing the functionality in the frontend GUI.

5.3 Creating API Models from API Traffic

This feature, as described in section 2.3, is currently partially implemented – the API traffic session functionality of the frontend and backend allows the user to store traffic sessions so that the system can analyze them at a later time, and to retrieve details about stored sessions. The frontend implements full CRUD functionality for traffic sessions. However, the system does not yet generate models from the stored traffic session.

Implementing this feature would involve creating an algorithm that converts stored API traffic sessions into either full API model revisions, or a list of modifications to an existing revision (thereby creating a new revision). The frontend would need to be extended to associate traffic sessions with API model revisions, and possibly provide a list of model changes that the user can selectively accept or deny.

5.4 Exposing API Modeling Service Functionality in Frontend GUI

The API Modeling Service has extra functionality that is currently not being exposed in the Modeler frontend. Particularly, while the Modeling Service can generate API documentation (see section 2.2), the frontend does not have a feature to view the generated documentation for a particular API revision.

Implementing this involves a fairly trivial change to the frontend to expose the link to the generated API documentation to the user.

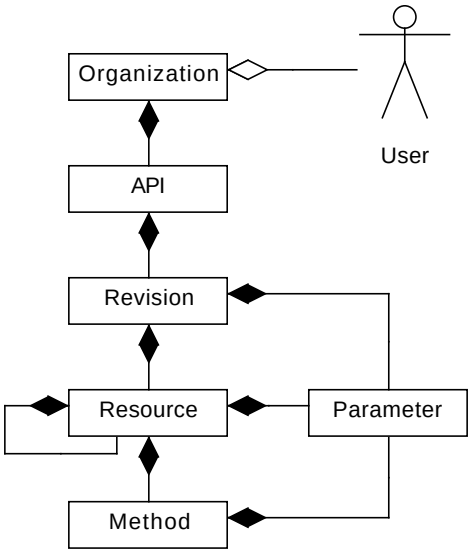

Figure 15: API Modeling Service architecture (simplified)

A API Modeling Service Overview

The API Modeling Service is an API that provides functions to create, edit, and store API models. In addition, the Service provides services that operate on API models, like generating API documentation (see section 2.2).

Figure 15 shows an overview of the basic architecture of the API Modeling Service – it is structured around organizations, APIs, resources, and methods. Each user of the Service belongs to one or more organizations. Each organization has zero or more APIs. Each API contains zero or more API revisions – the revision can be considered a discrete "API model." Each revision consists of zero or more resources, and resources contain zero or more methods. Revisions, resources, methods can contain zero or more parameters.

Consult the API Modeling V1 API Specification [18] for a comprehensive description of its API.

B Example API

This example API describes a simple service that stores to-do messages, and is used throughout this document to illustrate various concepts. The base URL is assumed to be `https://api.example.com/v1`. Error responses are omitted for the sake of brevity. The API response with HTTP response code 200 (OK) unless specified otherwise.

`/` root resource

`/todos` Collection of to-dos

GET returns a list of all to-dos

Response Body (e.g. `GET /todos`)

```
[
  { "id": "1", ... },
  { "id": "2", ... },
  ...
]
```

POST creates a new to-do

Request Body (e.g. `POST /todos`)

```
{
  "todo": "Finish senior project",
}
```

Response Body HTTP 201 (Created)

```
{
  "id": "123",
  "done": false,
  "todo": "Finish senior project"
}
```

The `id` field is auto-generated – the value of the `id` field in the request body is ignored, if specified. The `done` field defaults to `false`, if unspecified.

`/todos/{todoId}` A single to-do

GET returns the to-do

Response Body (e.g. `GET /todos/123`)

```
{
  "id": "123",
  "done": false,
  "todo": "Finish senior project"
}
```

PUT updates the to-do

Request Body (e.g. `PUT /todos/123`)

```
{
  "done": true
}
```

Response Body (e.g. `PUT /todos/123`)

```
{
  "id": "123",
  "done": true,
  "todo": "Finish senior project"
}
```

DELETE deletes the to-do

Response Body HTTP 204 (No Content)

Listing 1: Example API Model (in API Modeling Service JSON format)

```
{
    "revisionNumber": 1,
    "baseUrl": "http://api.example.com/v1",
    "description": "First version of the To-Dos API.",
    "displayName": "To-Dos Revision 1",
    "releaseVersion": "v1",
    "resources": [
        {
            "name": "todos_todoId",
            "displayName": "/todos/{todoId}",
            "description": "A single To-Do",
            "path": "/todos/{todoId}",
            "methods": [
                {
                    "name": "todos_todoId_GET",
                    "displayName": "GET /todos/{todoId}",
                    "description": "Return a To-Do",
                    "path": "/todos/{todoId}",
                    "verb": "GET"
                },
                {
                    "name": "todos_todoId_PUT",
                    "displayName": "PUT /todos/{todoId}",
                    "description": "Update a To-Do",
                    "path": "/todos/{todoId}",
                    "verb": "PUT"
                },
                {
                    "name": "todos_todoId_DELETE",
                    "displayName": "DELETE /todos/{todoId}",
                    "description": "Delete a To-Do",
                    "path": "/todos/{todoId}",
                    "verb": "DELETE"
                }
            ],
            "parameters": [
                {
                    "name": "todoId",
                    "description": "To-Do ID",
                    "dataType": "number",
                    "required": true,
                    "type": "template"
                }
```

```
            ]
        },
        {
            "name": "todos",
            "displayName": "/todos",
            "description": "Collection of all to-dos",
            "path": "/todos",
            "methods": [
                {
                    "name": "todos_GET",
                    "displayName": "GET /todos",
                    "description": "Return a list of all To-Dos",
                    "path": "/todos",
                    "verb": "GET"
                },
                {
                    "name": "todos_POST",
                    "displayName": "POST /todos",
                    "description": "Create a new To-Do",
                    "path": "/todos",
                    "verb": "POST"
                }
            ]
        }
    ]
}
```

Glossary

base URL

The root URL of all (or most) API requests. Paths are resolved relative to this URL. For example, requests for the resource `/posts/1/comments` with base URL `https://api.example.com/v1` are made to the final URL `https://api.example.com/v1/posts/1/comments`. Also referred to as base path, base URI, and endpoint.

client

The consumer of the API that makes requests and receives responses over HTTP.

collection

A type of resource that groups sub-resources. The client can typically query the collection to retrieve a list of sub-resources, retrieve a specific sub-resources by its identifier (e.g. `/posts/1` for the "first" resource in the "posts" collection), add and delete sub-resources.

header

A key-value pair that is sent in the header of an HTTP request to an API

method

A combination of a resource and verb that describes an API action or operation

parameter

A variable in the description of an API model. For example, a template parameter, header parameter, or query parameter. Models can specify parameters that are required to be present for each request.

resource

A location, represented as an HTTP path (e.g. `/posts/1/comments`), that can be manipulated with verbs.

revision

An API model that describes a specific version of the API

template parameter

A part of a resource path defined in an API model, typically denoted by the parameter name surrounded by curly braces, that can be substituted by the client with a value subject to restrictions defined in the API model. For example, in the resource path `/posts/{id}`, there is one template parameter named `id` that can be substituted to create a valid resource path: `/posts/123`

verb

A standard HTTP method (aka verb) that specifies an operation on a resource (e.g. `GET` to retrieve the resource data, or `PUT` to set it).

Acronyms

API Application Programming Interface

CRUD Create-Read-Update-Delete, a concept that encapsulates common operations when working with data

GUI graphical user interface

HAR HTTP Archive [19]

HTTP HyperText Transfer Protocol [1]

HTTPS HTTP Secure, i.e. HTTP Over TLS [2] or related protocol

ID identifier, a property of a resource inside a collection that identifies it uniquely at least within that collection

IDE integrated development environment

JSON JavaScript Object Notation [3]

REST Representational State Transfer [20]

UI user interface

URL Uniform Resource Locator

UUID Universally Unique Identifier

WADL Web Application Description Language [4]

WSDL Web Services Description Language [5]

References

[1] R. Fielding, J. Gettys, J. Mogul, et al. *Hypertext Transfer Protocol – HTTP/1.1*. Tech. rep. Internet Engineering Task Force, June 1999. URL: http://tools.ietf.org/html/rfc2616.

[2] E. Rescorla. *HTTP Over TLS*. Tech. rep. Internet Engineering Task Force, May 2000. URL: http://tools.ietf.org/html/rfc2818.

[3] D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON)*. Tech. rep. Internet Engineering Task Force, July 2006. URL: http://tools.ietf.org/html/rfc4627.

[4] Marc J. Hadley. *Web Application Description Language*. Tech. rep. World Wide Web Consortium, Aug. 2009. URL: http://www.w3.org/Submission/wadl/.

[5] Erik Christensen, Francisco Curbera, Greg Meredith, et al. *Web Service Definition Language (WSDL)*. Tech. rep. World Wide Web Consortium, 2001.

[6] Haxx. *curl and libcurl*. 2014. URL: http://curl.haxx.se/.

[7] Dimitri van Heesch. *Doxygen*. 2014. URL: http://doxygen.org.

[8] Python Software Foundation. *Welcome to Python.org*. 2014. URL: http://www.python.org/.

[9] Google. *AngularJS – Superheroic JavaScript MVW Framework*. 2013. URL: http://angularjs.org.

[10] Hampton Catlin et al. *Sass: Syntactically Awesome Style Sheets*. 2014. URL: http://sass-lang.com.

[11] Twitter Inc. *Bootstrap*. 2014. URL: http://getbootstrap.com.

[12] Trygve Reenskaug. *MVC XEROX PARC 1978-79*. 1979. URL: http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html.

[13] AngularUI Team. *AngularUI Router*. 2014. URL: https://github.com/angular-ui/ui-router.

[14] Joyent Inc. *node.js*. 2013. URL: http://nodejs.org.

[15] Thomas Steiner. *REST Describe & Compile*. 2007. URL: http://tomayac.de/rest-describe/latest/RestDescribe.html.

[16] Thomas Steiner. "Automatic Multi Language Program Library Generation for REST APIs". PhD thesis. University of Karlsruhe, Aug. 2007.

[17] Reverb Technologies Inc. *Swagger: A simple, open standard for describing REST APIs with JSON*. 2013. URL: https://helloreverb.com/developers/swagger.

[18] Vinoth Pethaiyan. *API Modelling V1 API Specification*. Tech. rep. Apigee, Inc., Dec. 2013.

[19] Jan Odvarko, Arvind Jain, and Andy Davies. *HTTP Archive (HAR) format*. Tech. rep. World Wide Web Consortium, Aug. 2012. URL: https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/HAR/Overview.html.

[20] Roy Thomas Fielding. "Architectural styles and the design of network-based software architectures". PhD thesis. University of California, Irvine, 2000.