D Web Development

This book guides you through all aspects of web development with D and the vibe.d framework. Covering today's popular operating systems, this guide starts with the setup of your development system. From the first Hello World-style application, you will move on to building static web pages with templates. Next, you will add database access to your application, providing persistent storage for your data. Learning about the internals of vibe.d, you will be able to use low-level techniques such as raw TCP access. The vibe.d concepts can also be used for GUI clients, which is the next topic that you will learn. This comprehensive guide concludes with an overview of the most useful vibe.d extensions and where to find them.

The concepts are always illustrated with source code, giving you an insight into how to apply them in your application.

Who this book is written for
Whether you are new to the world of D, or have already developed applications in D, or if you want to leverage the power of D for web development, then this book is ideal for you. Basic knowledge of core web technologies such as HTML5 is helpful but not required.

What you will learn from this book
- Create amazingly fast web applications with D
- Use Diet templates to easily create a web user interface
- Utilize the web framework for interactive applications with input validation and internationalization
- Access a database to provide persistent storage for your application
- Extend your application with a REST interface and access other applications via REST
- Understand vibe.d’s fiber-based approach to asynchronous I/O and use it for the integration of existing components
- Create GUI applications with vibe.d


Foreword by Andrei Alexandrescu - Co-developer of the D programming language

Kai Nacke
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 6 'Using the REST Interface'
- A synopsis of the book’s content
- More information on D Web Development
Kai Nacke is a professional IT architect living in Düsseldorf, Germany. He holds a diploma in computer science from the University of Dortmund. His diploma thesis about universal hash functions was recognized as the best of the semester. He has been with IBM for more than 15 years, and has great experience in the development and architecture of business and enterprise applications.

Fascinated by the first home computer, he learned to program a VIC-20 in BASIC. Later, he turned to Turbo PASCAL and Small C on CP/M. Experimenting with the source of Small C created his interest in compiler technology. Many computers, operating systems, and languages followed these first steps.

Around 2005, he became interested in the D programming language and created the first fun applications in D. Missing a 64-bit D compiler for Windows, he started to contribute to the LLVM compiler framework and LDC, the LLVM-based D compiler. Soon, he became committer of both projects and is now the current maintainer of LDC.

He is also a speaker at the Free and Open Source Software Developers' European Meeting (FOSDEM) and was one of the reviewers of *D Cookbook*, Packt Publishing.
Preface

In the cloud age, web technologies are more important than ever. The vibe.d framework enables you to use the D programming language for a wide range of web-related tasks. The D programming language allows elegant solutions for common problems, while native compilation produces fast binaries. The vibe.d framework takes advantage of these language features. Together with the innovative use of fibers, the applications that you build are scalable and have a very quick response time.

This book will explain everything you need to know about the vibe.d framework in order to successfully build and run web applications.

What this book covers

Chapter 1, Getting Started with Your First Web Application, explains how to set up and use your development environment. At the end of this chapter, you will have already created your first web application.

Chapter 2, Using Templates for Your Web Pages, covers the Diet template engine. You will learn all about templates—from creating simple static templates to using D code in templates.

Chapter 3, Get Interactive – Forms and Flow Control, brings web forms to your application and introduces route matching.

Chapter 4, Easy Forms with the Web Framework, discusses how to validate user input.

Chapter 5, Accessing a Database, shows how to use a database in an application using a variety of SQL and NoSQL bases.
Preface

Chapter 6, Using the REST Interface, teaches you about REST services. You will learn how to provide and consume a generated REST service. You will also study how to interface with an existing REST service.

Chapter 7, The vibe.d Internals, introduces you to the fiber-based pseudo-blocking programming model that is the base for vibe.d.

Chapter 8, Using vibe.d with a GUI Client, applies the vibe.d programming model to a graphical UI application.

Chapter 9, Power Your Application with vibe.d Extensions, shows you what other developers have already implemented with vibe.d and how to publish your application.
Using the REST Interface

Applications are usually built using components that can be distributed on the network. The protocol that is used to communicate with remote components should be fast but simple to use. Representational State Transfer (REST) is a software architecture style based on the principles of HTTP—the protocol of the World Wide Web (WWW). Combined with simple data representation, REST is a popular way to access remote components or services.

In this chapter, we will cover the following topics:

- What is REST and how it uses JSON
- How to expose an interface via REST
- How to access a component remotely via REST
- How to tailor the URL path and parameter passing

Defining the principles of the World Wide Web

As soon as computers were networked, people created binary protocols to execute code on a remote computer. The disadvantage of these Remote Procedure Call (RPC) protocols is the system dependency. Due to the binary nature, it is not easy to get a remote procedure call right: the size of data types, endianness, and alignment may be different on the target machine. With the success of the WWW, the idea of services appeared. A service provides some functionality, for example, retrieving the stock price at NASDAQ in real time. Simple Object Access Protocol (SOAP) was created. SOAP is based on XML and, therefore, enables structured data exchange in heterogeneous networks. Web Service Description Language (WSDL) was created to describe the services, again in XML.
The combination of SOAP and WSDL is used to look up services and generate language bindings for them. This works very well, but there are drawbacks. SOAP is designed to be independent of the transport layer. HTTP is often used but other protocols, including e-mail, are possible. Due to this, SOAP contains certain features that are also available with HTTP. Instead of encrypting the SOAP message, you can use an encrypted HTTP connection. Another drawback is that you lose flexibility because most tools generate code from the service description and cannot handle additions to the service description without rerunning the code generator.

Another approach is to utilize the success factors of the WWW. Uniform Resource Locator (URL) already describes a resource in a unique way. The HTTP method is an action on this resource: GET can be interpreted to retrieve data while POST creates some data, PUT updates the data, and DELETE deletes it. Combine this with a data representation such as JSON or XML and you have a simple service defined. REST is a software architecture based on this approach. Due to this, you will not find a REST standard. A software architecture is called REST if it adheres to the following constraints:

- There is a uniform interface between the client and server. This can be the HTTP communication described previously
- The communication is stateless
- The client can cache responses
- The system is layered, for example, a client does not know if it is directly connected to a server or through a proxy

If a system fulfills these constraints, it is called RESTful.

From the point of view of the system architecture, it is not an easy decision between SOAP/WSDL and REST because of the many different details. However, vibe.d currently supports REST only, which has been used in this chapter.

**Serializing D to JSON and back**

With the RESTful approach, the client and server exchange messages using the HTTP protocol. Of the many possible data formats, JavaScript Object Notation (JSON) is used. Originally, JSON was based on a subset of JavaScript. It has basic data types for numbers, strings, and Booleans. Like JavaScript, it uses two different structures to create complex data:

- A collection of key/value pairs called an object in JSON. This is like an associative array in D and can be used to represent objects, maps, and so on. The key/value pairs are surrounded by { and }.


• An ordered list of values called an array in JSON. The values are surrounded by [ and ].

The format is written in JavaScript. Once properly formatted, this is easily readable. A list of notes could look as follows:

```json
[ 
  { "id":1, "topic":"Topic 1", "content":"Content 1" },
  { "id":2, "topic":"Topic 2", "content":"Content 2" }
]
```

A single note is the key/value collection in { }. The list consists of the enumerated notes in [ ]. It is possible to construct arbitrarily complex data representations with this approach.

JSON is human readable. This is a big advantage while debugging, but the text representation is less compact and always requires the receiver to parse the message. A binary representation called Binary JSON (BSON) was developed to address these issues. BSON is supported by vibe.d as well. However, it is not used for REST but for the MongoDB protocol.

D types are serialized to JSON in a straightforward way. D basic types are mapped to the corresponding basic JSON type. D arrays are mapped to JSON arrays and D struct types, class types, and associative arrays are mapped to JSON objects.

A Json struct (declared in the vibe.data.json module) is used to hold a single JSON value. In order to serialize a note, you can write the following:

```d
Note note = { 1, "Topic 1", "Content 1" };  
auto json = serializeToJson(note);  
auto jsonString = json.toString();
```

The serializeToJsonString() function creates a string in one call. The serializeToPrettyJson() function creates a JSON string with white space and new lines are added for better readability.

To deserialize a JSON string, you can type the following:

```d
auto json =  
  parseJsonString(`{"id":1,"topic":"t1","content":"c1"}`);  
auto note = deserializeJson!Note(json);
You can change the serialization, too. You need to implement only one of the following function pairs in struct or class. The first one is as follows:

```cpp
Json toJson() const;
static T fromJson(Json src);
```

Here is the second one:

```cpp
string toString() const;
static T fromString(string src);
```

One use case is if you want to map an enumeration value to the string literal instead of the integer value. Let’s suppose that you have the following enumeration:

```cpp
enum Color { Red, Green, Blue }
```

The standard serialization maps `Color.Red` to 0, `Color.Green` to 1, and `Color.Blue` to 2. The following code maps the values to `Red`, `Green`, and `Blue` instead:

```cpp
struct ColorHolder
{
    import std.conv: to;
    Color color;

    const Json toJson()
    {
        return Json(color.to!string());
    }

    static ColorHolder fromJson(Json src)
    {
        return ColorHolder(src.deserializeJson!string().to!Color());
    }
}
```

### Creating and using a REST service

The `NoteStore` class is used to persist and retrieve notes. Because other applications may require this functionality, too, you want to provide it as a service. Starting with the implementation based on Redis from *Chapter 5, Accessing a Database*, you can extract the following interface and save it as `notestore.d`:

```cpp
module notestore;

struct Note
{
```
The only change here is that `addNote()` cannot have a `ref Note` parameter. The reason is that the caller of the method lives in a different process that is possible on a different computer. Therefore, the `Note` struct of the caller cannot be changed.

The server and client both require this interface. You use a separate module for it.

### Providing a service

Let's create the server side. The main difference from the previous implementations is that the implementation of the `NoteStore` interface is registered as a REST service via a call to the `registerRestInterface()` method:

1. Create a new project with `dub`:
   
   ```
   $ dub init noteapp_server --type=vibe.d
   ```

2. Copy the `notestore.d` module we just saved in the previous section to the `source` folder.

3. Implement the functionality in the `app.d` module in the `source` folder. The implementation is based on the Redis version from *Chapter 5, Accessing a Database*:

   ```
   import vibe.d;
   import notestore;
   import std.conv : to;
   import std.format : format;
   
   shared static this()
   {
     auto redis = new RedisClient();
   }
   
   auto noteStore = new
     NoteStoreImplementation(redis.getDatabase(0));
   
   auto router = new URLRouter;
   ```
Using the REST Interface

```
router.registerRestInterface(noteStore);

auto settings = new HTTPServerSettings;
settings.port = 8081;
settings.bindAddresses = [":1", "127.0.0.1"];
listenHTTP(settings, router);

logInfo("Please open http://127.0.0.1:8081/ in your browser.");
```

```cpp
class NoteStoreImplementation : NoteStore
{
    RedisDatabase db;

    this(RedisDatabase db)
    {
        this.db = db;
    }

    Note[] getNotes(string name)
    {
        import std.array : array;
        Note[] result = new Note[0];
        auto prefix = format("user:%s", name);
        auto noteskey = format("%s:notes", prefix);
        auto ids = db.lrange!long(noteskey, 0, db.llen(noteskey));
        foreach (id; ids)
        {
            auto key = format("%s:note:%d", prefix, id);
            auto data = array(db.hmget(key, "topic", "content"));
            if (!data.empty)
                result ~= Note(id, data[0], data[1]);
        }
        return result;
    }

    long addNote(string name, Note note)
    {
        auto prefix = format("user:%s", name);
        note.id = db.incr(format("%s:nextid", prefix));
```
With the call to `registerRestInterface()`, vibe.d adds routes that correspond with the names of the methods in the `NoteStore` interface. It uses the same algorithm as the web framework described in Chapter 4, *Easy Forms with the Web Framework*. The details of the path generation are explained here.

Start your Redis database, run this application, and open `http://127.0.0.1:8081/notes?name=yourid` in your browser. If the database still has the content from the previous chapters, then you can see a JSON representation of it in your browser.

[Note the changed port number!]

**Using a service**

Using the previously created service is simple, as follows:

1. Create a copy of the note application based on the Redis database from Chapter 5, *Accessing a Database*.
2. Copy the `notestore.d` module we just saved in the previous section to the source folder.
3. In the `app.d` module, you need to make the following changes:
   
   - Add an import `notestore;` statement at the top of the file.
   - Delete the `NoteStore` class and the `Note` struct because they are now imported.
   - Delete the `noteStore` variable and the static `this()` constructor that initializes this variable.
   - Add the following code to use a `NoteStore` service via REST:
     ```
     private __gshared RestInterfaceClient!NoteStore noteStore;
     shared static this()
     {
         noteStore = new RestInterfaceClient!NoteStore("http://127.0.0.1:8081/");
     }
     ```
     ```
Instantiating the `RestInterfaceClient` class generates a proxy class that forwards all the method calls to the remote service at http://127.0.0.1:8081.

To test your new application, you need to start the Redis database, server application from the previous section, and this client application.

**Tailoring the generated REST API**

REST is really a tool for service integration. To match the requirements of an existing service, you need control of several parameters such as the path, HTTP method, or header parameters.

**Changing the generated path**

As in the case of a web application, the URL path and HTTP method is derived from the function to be called.

A `@property` method uses the HTTP `GET` method to read the property and `PUT` to write the property. The HTTP method of all the other methods is derived from the prefix of the method name:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>HTTP method</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>GET</td>
</tr>
<tr>
<td>query</td>
<td>GET</td>
</tr>
<tr>
<td>set</td>
<td>PUT</td>
</tr>
<tr>
<td>put</td>
<td>PUT</td>
</tr>
<tr>
<td>update</td>
<td>PATCH</td>
</tr>
<tr>
<td>patch</td>
<td>PATCH</td>
</tr>
<tr>
<td>add</td>
<td>POST</td>
</tr>
<tr>
<td>create</td>
<td>POST</td>
</tr>
<tr>
<td>post</td>
<td>POST</td>
</tr>
<tr>
<td>remove</td>
<td>DELETE</td>
</tr>
<tr>
<td>erase</td>
<td>DELETE</td>
</tr>
<tr>
<td>delete</td>
<td>DELETE</td>
</tr>
</tbody>
</table>

The default HTTP `POST` method is used if the method is neither an `@property` method nor a method that can be derived from the preceding table. You can set the HTTP method with the `@method` annotation:

```java
@method(HTTPMethod.GET)
Note retrieveNote();
```
This overrides the default POST method with GET.

The URL path is created from the method name. First, the prefix, according to the preceding table, is removed. Then, the remaining string is mapped using the chosen naming scheme. The default naming scheme is MethodStyle.lowerUnderscored: an underscore is inserted in front of every upper character and then all the characters are made lowercase. For example, the getAllMyData() method name is mapped to all_my_data. The naming scheme is an optional parameter of the registerRestInterface() method. Just pass a different value if you want a different naming scheme. The following are the possible values:

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>camelCase</td>
<td>allMyData</td>
</tr>
<tr>
<td>lowercase</td>
<td>allmydata</td>
</tr>
<tr>
<td>lowerUnderscored</td>
<td>all_my_data</td>
</tr>
<tr>
<td>PascalCase</td>
<td>AllMyData</td>
</tr>
<tr>
<td>unaltered</td>
<td>getAllMyData</td>
</tr>
<tr>
<td>uppercase</td>
<td>ALLMYDATA</td>
</tr>
<tr>
<td>upperUnderscored</td>
<td>ALL_MY_DATA</td>
</tr>
</tbody>
</table>

You can override the name with the @path attribute. It is also possible to specify a relative path with @path:

```java
@path("note/all")
Note[] getAllNotes();
```

The getAllNotes() method is now called with GET /note/all instead of GET /all_notes.

A common URL prefix can be set with the @path annotation as follows:

```java
@path("notestore")
interface NoteStore
{
    long addNote(string name, Note note);
}
```

The addNote() method is now called with POST /notestore/note instead of POST /note.

As an alternative, you can use @rootPathFromName. Then the URL prefix is derived from the interface name. This is a shorthand for @path("").
Using the REST Interface

If the first parameter of a method is named \texttt{id}, then this parameter is mapped to the \texttt{void putData(string id, string data);} path using the \texttt{/:id/data} path. This is really a legacy mechanism but you must be aware of it in order to avoid surprises. There is also a more generalized approach. If a parameter name starts with an underscore, then this parameter is not serialized. Instead, it is available as a placeholder and can be used for path generation. Let's take a look at the following declaration:

\begin{verbatim}
@method (HTTMethod.GET)
@path(":_name")
void existsDatabase(string _name)
\end{verbatim}

This results in \texttt{GET /notestore} if the method is called with \texttt{_name} set to \texttt{notestore}. This flexibility to change the generated path is useful if you want to access an existing service.

Passing parameters

If a method of a REST interface is called, the parameters are passed as a query string if the HTTP \texttt{GET} or \texttt{HEAD} methods are used. Otherwise, they are serialized to JSON and transmitted in the request body. This behavior can be changed with the following annotations:

- With \texttt{@headerParam}, the parameter is passed in the HTTP header. This can be used to add new elements to the header, for example, an \texttt{If-Match} header:

\begin{verbatim}
@headerParam("rev", "If-Match")
void postData(string rev, string data)
\end{verbatim}

- With \texttt{@queryParam}, the parameter is passed in the query string. In the example, the value of the query parameter is appended as \texttt{?param=value of parameter query} to the request URL:

\begin{verbatim}
@queryParam("query", "param")
void postData(string query, string otherdata)
\end{verbatim}

- With \texttt{@bodyParam}, the parameter becomes part of the JSON object transmitted in the request body. A limitation here is that serialization in the JSON object is not customizable. This can be used to create request bodies if the HTTP \texttt{GET} method is used:

\begin{verbatim}
@bodyParam("data", "content")
void getData(string data)
\end{verbatim}
Accessing CouchDB

The NoteStore service and NoteApp client from the previous sections were written in D. Therefore, there was no problem in using the service. However, the main idea of REST is to be independent of the service implementation. This is best shown by using a different service.

CouchDB is another document-based database. It is written in the Erlang programming language and offers a REST interface that you can use to implement NoteStore.

Installing CouchDB

The website of CouchDB is http://couchdb.apache.org/. Here, you can find detailed installation instructions and precompiled binaries for OS X and Windows. Building from the source requires the Erlang programming language. For a quick start, you can use the version of your distribution:

- On Ubuntu/Debian, type the following to install CouchDB:
  $ sudo apt-get install couchdb
- On Fedora 21 or earlier, you can install CouchDB with the following:
  $ sudo yum install couchdb
- On Fedora 22, you type as follows:
  $ sudo dnf install couchdb
- On OS X, the Homebrew package manager can be used to install CouchDB:
  $ brew install couchdb

After the installation, you can start the CouchDB server with sudo /etc/init.d/couchdb. On OS X, you can start the CouchDB server with launchctl load /usr/local/Cellar/couchdb/1.6.1_3/homebrew.mxcl.couchdb.plist.

Testing the REST interface

CouchDB listens on port 5984 for the incoming HTTP requests.

The complete API is documented online at http://docs.couchdb.org/en/1.6.1/api/index.html.
The vibe.d framework includes an HTTP client class that is used by the generated REST client. You need to make the decision if you want to use RestInterfaceClient or if you have to create your own access class using the HTTP client. If you scan through the online documentation of CouchDB, then you can note the following deficiencies of RestInterfaceClient:

- You have no access to the HTTP status code for successful requests. If the request fails, then the HTTP status code is available in RestException thrown by RestInterfaceClient.
- RestInterfaceClient requires a non-empty response body. However, CouchDB provides you with some check APIs that only return an HTTP status code and empty body.

For a full-features interface to CouchDB, these restrictions may not be acceptable. However, to implement NoteStore, only some API calls are required. The decision is to go with RestInterfaceClient because it reduces the effort to implement the required functionality.

To develop the interface class, it is helpful to use a small test application. Create a new project with the following:

```
$ dub init couchdb --type=vibe.d
```

Then replace VibeDefaultMain with VibeCustomMain in the dub.sdl file. Now you can use a main() method.

Another tool that you can use is the logging infrastructure. Every example application outputs a short message using logInfo(). The framework itself uses different logging levels to output diagnostic messages. The default is to output only informational messages, warnings, and errors. This can be changed with the setLogLevel() method. Setting the level to LogLevel.verbose2 outputs the request URLs and the request and response bodies.

Let's try to get some information about CouchDB and create a database. A GET request to the base URL returns information about CouchDB. A GET request with a single path element returns information about the database with the name of the path element or status code 404 if the database does not exist. A PUT request with a single path element creates the database or returns a status code 412 if the database already exists.

Remember that RestInterfaceClient throws RestException if the HTTP status code does not indicate a successful execution!
With this API information, you can write your first CouchDB client that you can put in the `app.d` module of the previously created project:

```d
import vibe.d;

interface CouchDB
{
    Json get();

    @method(HTTPMethod.GET)
    @path("/:db")
    Json existsDB(string _db);

    @method(HTTPMethod.PUT)
    @path("/:db")
    Json createDB(string _db);
}

void main()
{
    // Uncomment to see the requests
    // setLogLevel(LogLevel.verbose2);
    auto couchdb = new
        RestInterfaceClient!CouchDB("http://127.0.0.1:5984/");

    auto ver = couchdb.get();
    logInfo("CouchDB: %s", ver["couchdb"].to!string);
    logInfo("Version: %s", ver["version"].to!string);
    logInfo("Vendor-Version: %s",
        ver["vendor"]["version"].to!string);
    logInfo("Vendor-Name: %s", ver["vendor"]["name"].to!string);
    logInfo("UUID: %s", ver["uuid"].to!string);

    try
    {
        couchdb.existsDB("notestore");
        logInfo("Database exists");
    }
    catch (RestException e)
    {
        logInfo("Database does not exist (HTTP status code %d)",
            e.status);

        couchdb.createDB("notestore");
    }
}
The CouchDB interface uses the JSON structure as the return type. It contains the parsed JSON string. You need to use this if the structure of the JSON string changes dynamically and cannot be mapped to a D structure. In this case, it is simply saved to define a D structure for the deserialization.

The get() method does not have a good name; getWelcome() would be better. The point here is that according to the rules, the get prefix is stripped from the method name, which results in an empty name. This results in a GET request to the base URL. There is no other way to specify this. The @path("/") or @path("") annotations are not allowed.

Implementing the NoteStore service

Encouraged by the success of the sample application, you can now implement the NoteStore class with CouchDB as a persistent store. The CouchDB API for documents is very similar to that for the CouchDB database. The first path element is again the database and the second path element is the document name.

As a special feature, CouchDB uses optimistic locking. Each document has a revision. If you want to update a document, you have to specify the revision of the document as well. If the current revision of the document in the database does not match the revision that you have provided, then somebody else has modified the document, your update conflicts with the current state of the database and this is indicated by the HTTP status code 409 (HTTPStatus.conflict). In this case, you have to retry your update. As this is a potentially infinite process, you have to use a loop.

Together, this leads to the following implementation of the NoteStore class. At first, you import the required modules and define the CouchDB interface with the needed database methods. This interface makes use of many of the annotations introduced earlier in this chapter:

```
import vibe.d;
import notestore;

interface CouchDB
{
  Json get();

  @method(HTTPMethod.GET)
  @path(":name")
  Json existsDB(string _name);

  @method(HTTPMethod.PUT)
  @path(":\db")
```
Json createDB(string _db);

@method(HTTPMethod.PUT)
@path("/:db/:docid")
@headerParam("rev", "If-Match")
void updateDoc(string _db, string _docid, string rev, Json doc);

@method(HTTPMethod.PUT)
@path("/:db/:docid")
void createDoc(string _db, string _docid, Json doc);

@method(HTTPMethod.DELETE)
@path("/:db/:docid")
@headerParam("rev", "If-Match")
void deleteDoc(string _db, string _docid, string rev);

@method(HTTPMethod.GET)
@path("/:db/:docid")
Json retrieveDoc(string _db, string _docid, bool latest = false);

Next, you define the static this() constructor. A REST client for CouchDB is instantiated. This CouchDB instance is used by the NoteStore implementation. The NoteStore instance is then exported as a REST interface:

shared static this()
{
    auto couchdb = new RestInterfaceClient!CouchDB("http://127.0.0.1:5984/");

    auto noteStore = new NoteStoreImplementation(couchdb);

    auto router = new URLRouter;
    router.registerRestInterface(noteStore);

    auto settings = new HTTPServerSettings;
    settings.port = 8081;
    settings.bindAddresses = ["::1", "127.0.0.1"];
    listenHTTP(settings, router);

    logInfo("Please open http://127.0.0.1:8081/ in your browser.");
}
Finally, you implement the NoteStore interface using CouchDB. The implementation is straightforward except for one detail. CouchDB uses optimistic locking. If you try to update a document and the version of the document does not match, then an exception is thrown. This happens if someone else has already updated the document. The solution is to retry the operation with the updated document. Another detail to notice is that the client is not responsible for providing the id of a new Note. Therefore, it is necessary to figure out which id to use:

```java
class NoteStoreImplementation : NoteStore
{
    RestInterfaceClient!CouchDB db;

    this(RestInterfaceClient!CouchDB db)
    {
        this.db = db;
    }

    Note[] getNotes(string name)
    {
        try
        {
            auto doc = db.retrieveDoc("notestore", name, true);
            return deserializeJson!(Note[])(doc["doc"]);
        }
        catch (RestException e)
        {
            if (e.status == HTTPStatus.notFound)
                return new Note[0];
            throw e;
        }
    }

    long addNote(string name, Note note)
    {
        while (true)
        {
            try
            {
                try
                {
                    auto doc = db.retrieveDoc("notestore", name, true);
                    Note[] notes;
                    if (doc["doc"].type() != Json.Type.undefined)
                        notes = deserializeJson!(Note[])(doc["doc"]);
                }
                catch (RestException e)
                {
                    if (e.status == HTTPStatus.concurrency)
                        return e.code();
                    throw e;
                }
            }
            catch (RestException e)
            {
                if (e.status == HTTPStatus.notFound)
                    return new Note[0];
                throw e;
            }
        }
    }
}
else
    notes = new Note[0];
long id = 0;
foreach (n; notes) id = max(id, n.id);

    note.id = id+1;
    notes ~= note;

    db.updateDoc("notestore", name,
        doc["_rev"].toString,
        serializeToJson(notes));
}
catch (RestException e)
{
    if (e.status != HTTPStatus.notFound) throw e;
    Note[] notes = new Note[1];
    note.id = 1;
    notes[0] = note;

    db.createDoc("notestore", name,
        serializeToJson(notes));
}
return note.id;
}
catch (RestException e)
{
    if (e.status != HTTPStatus.conflict) throw e;
}
}
}

This implementation can be used as a replacement for the NoteStore service based on the Redis database from the Creating and using a REST service section. The structure of the application that you have now created is as follows:

• The web application listens on port 8080 for requests from web browsers. It uses the Redis database to store usernames and their passwords.
• The Redis database uses a proprietary protocol and listens on port 6379.
• The NoteStore service provides persistence for notes. It provides a REST interface and listens on port 8081 for the incoming requests.
• Finally, a note is stored in CouchDB. This is again an application that provides a REST interface and listens on port 5984.
Using the REST Interface

All these servers can be installed on different nodes on the Internet: this is a distributed application! This application adheres to the REST principle mentioned in the Defining the principles of the World Wide Web section, and it is a RESTful application!

Summary

In this chapter, RESTful systems were introduced. The HTTP protocol is used for lightweight remote access using JSON objects as the data format. You learned how to create and consume a REST service with vibe.d. You also learned how to tailor the REST request, which is very useful if you have to match an existing interface. For testing purposes, you created an application that used its own main() method.

The difference between the main() method provided by vibe.d and your own main() method and the other internals of vibe.d are the topics of the next chapter.
Where to buy this book

You can buy D Web Development from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.