REST Fundamentals and Structure of HTML5 Document

REST FUNDAMENTALS

1. What is REST?

REST (Representational State Transfer) is an **architectural style** (not a protocol or a standard) for designing networked applications, introduced by Roy Fielding in his 2000 doctoral dissertation. A web service that follows REST principles is called a **RESTful API**. REST APIs almost always use **HTTP** and usually exchange data as **JSON**.

Core idea: everything is a **resource** (a user, a product, an order). Each resource has a unique URL, and the client uses standard HTTP methods to work with a **representation** of that resource (for example, a JSON document).

```
Client  --- GET /users/5 --------------->  Server
Client  <-- 200 OK + JSON of user 5 -----  Server
```

API = Application Programming Interface, a way for programs to talk to each other. **Web API** = an API accessed over HTTP.

2. The Six REST Constraints

Constraint	Meaning
1. Client-Server	UI (client) and data storage (server) are separated, so each can evolve independently
2. Stateless	Every request carries all information needed to process it; the server stores **no client session state** between requests
3. Cacheable	Responses must say whether they can be cached (`Cache-Control`, `ETag`) to improve performance
4. Uniform Interface	One consistent way to interact: resource URLs, standard methods, representations, self-descriptive messages
5. Layered System	Client cannot tell if it talks to the real server or an intermediary (load balancer, proxy, gateway)
6. Code on Demand (optional)	Server may send executable code (e.g. JavaScript) to the client

Uniform interface has four parts: identification of resources (URIs), manipulation through representations, self-descriptive messages, and **HATEOAS** (Hypermedia As The Engine Of Application State: responses include links telling the client what it can do next).

Stateless in practice: instead of server-side sessions, the client sends a token (e.g. `Authorization: Bearer <JWT>`) on every request. This makes servers easy to scale horizontally.

3. Resources and URI Design

A **resource** is a noun. A **URI** identifies it. Good REST URLs use **nouns, not verbs**, are **plural**, lowercase, and hierarchical.

Bad (RPC style)	Good (REST style)
`GET /getAllUsers`	`GET /users`
`POST /createUser`	`POST /users`
`GET /getUser?id=5`	`GET /users/5`
`POST /deleteUser/5`	`DELETE /users/5`
`GET /getOrdersOfUser/5`	`GET /users/5/orders`

Terms:

- **Collection**: `/users` (a list of resources)
- **Single resource (item)**: `/users/5`
- **Sub-resource**: `/users/5/orders/12`
- **Path parameter**: the `5` in `/users/5` (identifies a resource)
- **Query parameter**: `/users?role=admin&page=2&limit=10` (filtering, sorting, pagination, searching)

Guidelines: avoid deep nesting beyond two levels, avoid file extensions in URLs, use hyphens in multi-word names (`/blog-posts`), and version the API (`/api/v1/users`).

4. HTTP Methods and CRUD Mapping

CRUD	HTTP method	Example	Success code
Create	POST	`POST /users`	201 Created
Read (all)	GET	`GET /users`	200 OK
Read (one)	GET	`GET /users/5`	200 OK
Update (replace)	PUT	`PUT /users/5`	200 OK or 204 No Content
Update (partial)	PATCH	`PATCH /users/5`	200 OK
Delete	DELETE	`DELETE /users/5`	204 No Content

- **Safe** methods do not change server state: GET, HEAD, OPTIONS.
- **Idempotent** methods give the same result if repeated: GET, PUT, DELETE, HEAD, OPTIONS. POST is **not** idempotent (sending it twice creates two resources).
- **PUT vs PATCH**: PUT replaces the entire resource; PATCH changes only the fields sent.

5. Status Codes in REST

Code	Meaning	When to use
200 OK	Success	GET, PUT, PATCH
201 Created	Resource created	POST (also send `Location` header)
204 No Content	Success, no body	DELETE, PUT
400 Bad Request	Malformed or invalid input	Validation failure
401 Unauthorized	Not authenticated	Missing or bad token
403 Forbidden	Authenticated but not allowed	Insufficient permissions
404 Not Found	Resource does not exist	Unknown ID
405 Method Not Allowed	Method unsupported on this URL	`DELETE /users` when not allowed
409 Conflict	State conflict	Duplicate email
422 Unprocessable Entity	Semantic validation error	Valid JSON, wrong values
429 Too Many Requests	Rate limit exceeded	Throttling
500 Internal Server Error	Server bug	Unexpected failure
503 Service Unavailable	Server overloaded or down	Maintenance

6. Request and Response Format

REST is format-agnostic, but **JSON** is the standard. Content negotiation uses headers:

- `Content-Type`: format of the body being sent (`application/json`)
- `Accept`: format the client wants back (`application/json`)

Example request

```
POST /api/v1/users HTTP/1.1
Host: example.com
Content-Type: application/json
Authorization: Bearer eyJhbGciOi...

{
  'name': 'Asha',
  'email': 'asha@example.com'
}
```

(In real JSON, keys and strings use double quotes.)

Example response

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /api/v1/users/6

{ id: 6, name: Asha, email: asha@example.com }
```

Good error response design: return a proper status code plus a helpful body, such as a machine-readable code and a human message, for example `{ error: VALIDATION_ERROR, message: Email is required }`.

7. Other Important REST Topics

- **Pagination**: `GET /users?page=2&limit=10` (or cursor-based) so large lists are returned in chunks.
- **Filtering and sorting**: `GET /products?category=books&sort=-price`.
- **Versioning**: URL (`/v1/`), header (`Accept: application/vnd.myapp.v1+json`), or query parameter.
- **Authentication**: API keys, Basic Auth, **JWT** (JSON Web Token), **OAuth 2.0**. Always use HTTPS.
- **Authorization**: what the authenticated user may do (roles, scopes).
- **CORS** (Cross-Origin Resource Sharing): browser security rule; the server sends `Access-Control-Allow-Origin` to permit requests from other domains.
- **Rate limiting**: limit requests per client (429 status).
- **Idempotency keys**: let clients safely retry POST requests.
- **Documentation**: **OpenAPI/Swagger** describes endpoints in a standard format.
- **Richardson Maturity Model**: Level 0 (single endpoint, RPC style), Level 1 (resources), Level 2 (HTTP verbs + status codes), Level 3 (HATEOAS). Most real APIs are Level 2.

8. REST vs SOAP vs GraphQL

	REST	SOAP	GraphQL
Type	Architectural style	Protocol	Query language
Format	JSON (mostly), XML	XML only	JSON
Endpoints	Many (one per resource)	Single	Single (`/graphql`)
Over-fetching	Possible	Possible	Client picks exact fields
Caching	Easy (HTTP caching)	Hard	Harder
Best for	Public web APIs, CRUD	Enterprise, strict contracts	Complex, nested data needs

9. Programs for REST

Program 1: Complete REST API with Flask (`app.py`)

Install: `pip install flask`

```
from flask import Flask, jsonify, request, abort

app = Flask(__name__)

# In-memory 'database'
users = [
    {'id': 1, 'name': 'Asha', 'email': 'asha@example.com'},
    {'id': 2, 'name': 'Ravi', 'email': 'ravi@example.com'},
]
next_id = 3

def find_user(user_id):
    return next((u for u in users if u['id'] == user_id), None)

# READ all (with optional filtering and pagination)
@app.get('/api/v1/users')
def get_users():
    name = request.args.get('name')
    page = int(request.args.get('page', 1))
    limit = int(request.args.get('limit', 10))
    result = [u for u in users if not name or name.lower() in u['name'].lower()]
    start = (page - 1) * limit
    return jsonify(result[start:start + limit]), 200

# READ one
@app.get('/api/v1/users/<int:user_id>')
def get_user(user_id):
    user = find_user(user_id)
    if user is None:
        return jsonify({'error': 'NOT_FOUND', 'message': 'User not found'}), 404
    return jsonify(user), 200

# CREATE
@app.post('/api/v1/users')
def create_user():
    global next_id
    data = request.get_json(silent=True)
    if not data or 'name' not in data or 'email' not in data:
        return jsonify({'error': 'VALIDATION_ERROR', 'message': 'name and email are
required'}), 400
    if any(u['email'] == data['email'] for u in users):
        return jsonify({'error': 'CONFLICT', 'message': 'Email already exists'}), 409
    user = {'id': next_id, 'name': data['name'], 'email': data['email']}
    next_id += 1
    users.append(user)
    response = jsonify(user)
    response.status_code = 201
    response.headers['Location'] = f'/api/v1/users/{user["id"]}'
    return response

# UPDATE (replace)
@app.put('/api/v1/users/<int:user_id>')
def replace_user(user_id):
    user = find_user(user_id)
    if user is None:
```

```
        return jsonify({'error': 'NOT_FOUND'}), 404
    data = request.get_json()
    if not data or 'name' not in data or 'email' not in data:
        return jsonify({'error': 'VALIDATION_ERROR'}), 400
    user['name'], user['email'] = data['name'], data['email']
    return jsonify(user), 200

# UPDATE (partial)
@app.patch('/api/v1/users/<int:user_id>')
def update_user(user_id):
    user = find_user(user_id)
    if user is None:
        return jsonify({'error': 'NOT_FOUND'}), 404
    data = request.get_json() or {}
    user.update({k: v for k, v in data.items() if k in ('name', 'email')})
    return jsonify(user), 200

# DELETE
@app.delete('/api/v1/users/<int:user_id>')
def delete_user(user_id):
    user = find_user(user_id)
    if user is None:
        return jsonify({'error': 'NOT_FOUND'}), 404
    users.remove(user)
    return '', 204

if __name__ == '__main__':
    app.run(port=5000, debug=True)
```

Run with `python app.py`.

Program 2: Testing the API with curl

```
curl http://127.0.0.1:5000/api/v1/users
curl http://127.0.0.1:5000/api/v1/users/1
curl -X POST http://127.0.0.1:5000/api/v1/users -H 'Content-Type: application/json' -
d '{"name":"Meena","email":"meena@example.com"}'
curl -X PATCH http://127.0.0.1:5000/api/v1/users/1 -H 'Content-Type:
application/json' -d '{"name":"Asha K"}'
curl -X DELETE http://127.0.0.1:5000/api/v1/users/2 -i
```

Program 3: Consuming the API with Python `requests`

```
import requests

BASE = 'http://127.0.0.1:5000/api/v1'

r = requests.post(f'{BASE}/users', json={'name': 'Meena', 'email':
'meena@example.com'})
print(r.status_code, r.headers.get('Location'), r.json())

user_id = r.json()['id']
print(requests.get(f'{BASE}/users/{user_id}').json())
print(requests.patch(f'{BASE}/users/{user_id}', json={'name': 'Meena S'}).json())
print(requests.delete(f'{BASE}/users/{user_id}').status_code)   # 204
print(requests.get(f'{BASE}/users/{user_id}').status_code)      # 404
```

Program 4: Consuming the API with JavaScript `fetch`

```
const BASE = 'http://127.0.0.1:5000/api/v1';

async function demo() {
  // CREATE
  let res = await fetch(`${BASE}/users`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ name: 'Kiran', email: 'kiran@example.com' })
  });
  const created = await res.json();
  console.log(res.status, created);

  // READ
  res = await fetch(`${BASE}/users/${created.id}`);
  console.log(await res.json());

  // DELETE
  res = await fetch(`${BASE}/users/${created.id}`, { method: 'DELETE' });
  console.log(res.status);   // 204
}
demo().catch(console.error);
```

Note: calling from a web page on another origin needs CORS enabled on the server (`pip install flask-cors`, then `CORS(app)`).

Program 5: REST API with Node.js and Express

Install: `npm install express`

```
const express = require('express');
const app = express();
app.use(express.json());           // parse JSON request bodies

let users = [{ id: 1, name: 'Asha' }];
let nextId = 2;

app.get('/api/v1/users', (req, res) => res.json(users));

app.get('/api/v1/users/:id', (req, res) => {
  const user = users.find(u => u.id === Number(req.params.id));
  if (!user) return res.status(404).json({ error: 'NOT_FOUND' });
  res.json(user);
});

app.post('/api/v1/users', (req, res) => {
  if (!req.body.name) return res.status(400).json({ error: 'name is required' });
  const user = { id: nextId++, name: req.body.name };
  users.push(user);
  res.status(201).location(`/api/v1/users/${user.id}`).json(user);
});

app.delete('/api/v1/users/:id', (req, res) => {
  users = users.filter(u => u.id !== Number(req.params.id));
  res.status(204).end();
```

```
});

app.listen(3000, () => console.log('API on http://localhost:3000'));
```

Program 6: Token authentication sketch (stateless)

```
from flask import request, jsonify
from functools import wraps

VALID_TOKENS = {'secret-token-123': 'asha'}

def require_token(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        header = request.headers.get('Authorization', '')
        token = header.replace('Bearer ', '')
        if token not in VALID_TOKENS:
            return jsonify({'error': 'UNAUTHORIZED'}), 401
        return fn(*args, **kwargs)
    return wrapper

# usage:  @app.get('/api/v1/secret')  then  @require_token  above the function
```

Test: `curl -H 'Authorization: Bearer secret-token-123' http://127.0.0.1:5000/api/v1/secret`. Real systems use signed JWTs or OAuth 2.0.

STRUCTURE OF HTML5 DOCUMENTS

11. What is HTML5?

HTML (HyperText Markup Language) describes the **structure and meaning** of web content using **elements** made of **tags**. **HTML5** is the current major version (a living standard maintained by WHATWG). It added semantic elements, native audio/video, canvas, form enhancements, and APIs such as local storage and geolocation, and it removed the need for plugins like Flash.

A web page is built from three technologies: **HTML** (structure), **CSS** (presentation), **JavaScript** (behaviour).

12. Anatomy of an Element

```
<a href='https://example.com' target='_blank'>Visit site</a>
```

- `<a>` is the **opening tag**, `</a>` is the **closing tag**.
- `href` and `target` are **attributes** (name and value pairs, placed in the opening tag).
- `Visit site` is the **content**.
- Opening tag + content + closing tag = an **element**.
- **Void (empty) elements** have no content or closing tag: `<br>`, `<hr>`, `<img>`, `<input>`, `<meta>`, `<link>`.
- Elements can be **nested** to form a tree called the **DOM** (Document Object Model).

13. Basic Structure of an HTML5 Document

```
<!DOCTYPE html>
<html lang='en'>
<head>
  <meta charset='UTF-8'>
  <meta name='viewport' content='width=device-width, initial-scale=1.0'>
  <meta name='description' content='A short description of this page'>
  <title>My First HTML5 Page</title>
  <link rel='stylesheet' href='style.css'>
  <link rel='icon' href='favicon.ico'>
</head>
<body>
  <h1>Hello, HTML5!</h1>
  <p>This is my first page.</p>
  <script src='app.js'></script>
</body>
</html>
```

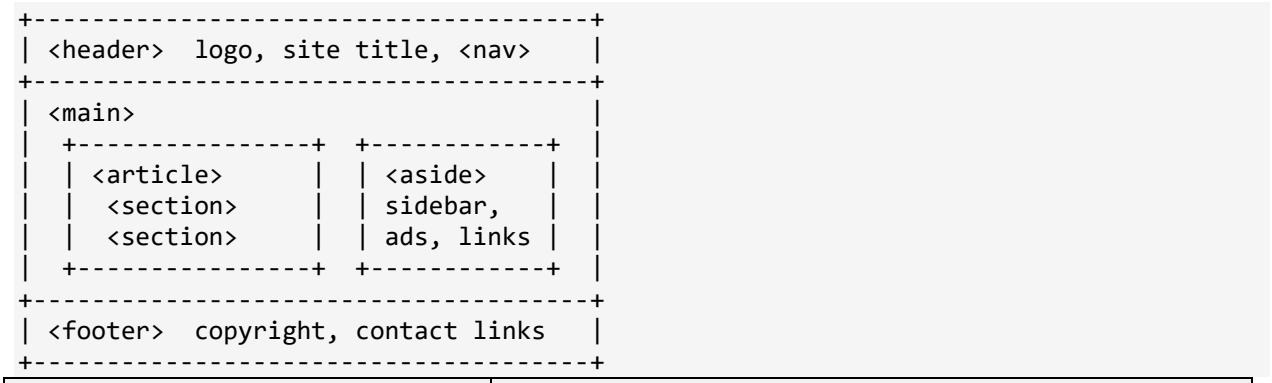
Explanation of each part:

Part	Purpose
`<!DOCTYPE html>`	Tells the browser this is an HTML5 document (triggers standards mode). Not a tag; it must be the very first line. Simpler than the long HTML 4 doctype.
`<html lang='en'>`	Root element that wraps everything. `lang` helps screen readers, translation and search engines.
`<head>`	Metadata: information **about** the page, not shown in the page body.
`<meta charset='UTF-8'>`	Character encoding so all languages and symbols display correctly. Place it early.
`<meta name='viewport'>`	Makes the page scale properly on mobile devices (responsive design).
`<meta name='description'>`	Summary shown by search engines.
`<title>`	Text in the browser tab and in search results.
`<link>`	Connects external resources such as CSS files and icons.
`<body>`	All visible content.
`<script>`	JavaScript. Placed at the end of `<body>` (or use `defer` in the head) so the page loads first.

Other head elements: `<style>` (internal CSS), `<base>` (base URL), `<meta name='author'>`, Open Graph tags for social sharing.

14. Semantic Elements of HTML5

Before HTML5, developers used `<div id='header'>` for everything. HTML5 introduced **semantic elements** whose names describe their meaning. Benefits: better **accessibility** (screen readers), better **SEO**, cleaner and more readable code.



Element	Purpose
`<header>`	Introductory content of a page or section (logo, title, navigation)
`<nav>`	Major navigation links
`<main>`	The unique, central content of the page (only **one** per page)
`<section>`	A thematic grouping of content, usually with a heading
`<article>`	Self-contained content that could stand alone (blog post, news story, comment)
`<aside>`	Related side content (sidebar, pull quote, advertisement)
`<footer>`	Footer of a page or section (copyright, contacts)
`<figure>` and `<figcaption>`	Image, diagram or code with a caption
`<time>`	Machine-readable date or time
`<mark>`	Highlighted text
`<address>`	Contact information
`<details>` and `<summary>`	Expandable show/hide widget

Section vs article vs div: use `<article>` when the content makes sense independently; `<section>` for a themed chunk of a larger document; `<div>` only when no semantic element fits (a generic container for styling).

15. Common Content Elements

Headings and text

```
<h1>Main heading (one per page)</h1>
<h2>Sub heading</h2> ... <h6>Smallest heading</h6>
<p>A paragraph with <strong>important</strong> and <em>emphasised</em> text.</p>
<br>   <!-- line break -->
<hr>   <!-- horizontal rule -->
<blockquote cite='https://example.com'>A long quotation.</blockquote>
<pre>Preformatted   text keeps   spacing</pre>
<code>inline code</code>
```

Prefer `<strong>` and `<em>` (meaning) over `<b>` and `<i>` (just appearance). Headings should follow order (do not skip from h1 to h4).

Links

```
<a href='https://example.com'>External link</a>
<a href='about.html'>Relative link</a>
<a href='#contact'>Jump to section with id=contact</a>
<a href='mailto:me@example.com'>Email me</a>
<a href='report.pdf' download>Download</a>
```

Images

```
<img src='images/photo.jpg' alt='A person reading a book' width='400' height='300'>

<figure>
  <img src='chart.png' alt='Sales chart for 2025'>
  <figcaption>Figure 1: Sales in 2025</figcaption>
</figure>
```

The `alt` attribute is essential for accessibility and when images fail to load.

Lists

```
<ul>   <!-- unordered -->
  <li>Apple</li>
  <li>Banana</li>
</ul>

<ol>   <!-- ordered -->
  <li>Step one</li>
  <li>Step two</li>
</ol>

<dl>   <!-- description list -->
  <dt>HTTP</dt>
  <dd>HyperText Transfer Protocol</dd>
</dl>
```

Tables

```
<table border='1'>
  <caption>Student Marks</caption>
  <thead>
    <tr><th>Name</th><th>Marks</th></tr>
  </thead>
  <tbody>
    <tr><td>Asha</td><td>92</td></tr>
    <tr><td>Ravi</td><td>85</td></tr>
  </tbody>
```

```
  <tfoot>
    <tr><td>Average</td><td>88.5</td></tr>
  </tfoot>
</table>
```

Use tables for tabular data only, never for page layout (use CSS Flexbox or Grid). `colspan` and `rowspan` merge cells.

Multimedia (new in HTML5)

```
<video controls width='480' poster='thumb.jpg'>
  <source src='movie.mp4' type='video/mp4'>
  <source src='movie.webm' type='video/webm'>
  Your browser does not support video.
</video>

<audio controls>
  <source src='song.mp3' type='audio/mpeg'>
</audio>
```

16. Forms and HTML5 Input Types

```
<form action='/api/v1/users' method='post'>
  <label for='name'>Name</label>
  <input type='text' id='name' name='name' required placeholder='Your name'>

  <label for='email'>Email</label>
  <input type='email' id='email' name='email' required>

  <label for='pwd'>Password</label>
  <input type='password' id='pwd' name='pwd' minlength='8' required>

  <label for='dob'>Date of birth</label>
  <input type='date' id='dob' name='dob'>

  <label for='age'>Age</label>
  <input type='number' id='age' name='age' min='1' max='120'>

  <label for='site'>Website</label>
  <input type='url' id='site' name='site'>

  <label for='phone'>Phone</label>
  <input type='tel' id='phone' name='phone' pattern='[0-9]{10}'>

  <label for='color'>Colour</label>
  <input type='color' id='color' name='color'>

  <label for='vol'>Volume</label>
  <input type='range' id='vol' name='vol' min='0' max='100'>

  <label for='city'>City</label>
  <select id='city' name='city'>
    <option value='cbe'>Coimbatore</option>
    <option value='chn'>Chennai</option>
  </select>
```

```
  <fieldset>
    <legend>Gender</legend>
    <input type='radio' id='f' name='gender' value='f'><label for='f'>Female</label>
    <input type='radio' id='m' name='gender' value='m'><label for='m'>Male</label>
  </fieldset>

  <input type='checkbox' id='terms' name='terms' required>
  <label for='terms'>I accept the terms</label>

  <label for='msg'>Message</label>
  <textarea id='msg' name='msg' rows='4' cols='40'></textarea>

  <button type='submit'>Register</button>
  <button type='reset'>Clear</button>
</form>
```

HTML5 form features: new input types (`email`, `url`, `date`, `number`, `range`, `color`, `tel`, `search`), attributes `required`, `placeholder`, `pattern`, `min`/`max`, `autofocus`, `autocomplete`, and built-in **client-side validation**. The `<label for>` pairing improves accessibility. `method='get'` puts data in the URL, `method='post'` puts it in the request body. (Browser forms only send GET and POST; to use PUT/DELETE, use `fetch` in JavaScript.)

17. Block vs Inline Elements

Block-level	Inline
Starts on a new line, takes full width	Stays within the line, takes only needed width
`<div>`, `<p>`, `<h1>` to `<h6>`, `<ul>`, `<section>`, `<table>`	`<span>`, `<a>`, `<strong>`, `<em>`, `<img>`, `<input>`

`<div>` is a generic block container; `<span>` is a generic inline container.

18. Global Attributes (work on any element)

Attribute	Use
`id`	Unique identifier (CSS, JS, link anchors)
`class`	One or more class names for CSS and JS (not unique)
`style`	Inline CSS
`title`	Tooltip text
`hidden`	Hides the element
`lang`	Language of the content
`data-*`	Custom data, e.g. `data-user-id='5'`
`tabindex`	Keyboard focus order
`contenteditable`	Makes content editable

Attribute	Use
`aria-*` and `role`	Accessibility information

Example: A Semantic HTML5 Page

Save as `index.html` and open it in a browser.

```
<!DOCTYPE html>
<html lang='en'>
<head>
  <meta charset='UTF-8'>
  <meta name='viewport' content='width=device-width, initial-scale=1.0'>
  <meta name='description' content='Demo page showing HTML5 structure'>
  <title>My Tech Blog</title>
  <style>
    * { box-sizing: border-box; }
    body { font-family: Arial, sans-serif; margin: 0; line-height: 1.6; color: #222;
}
    header, footer { background: #1f3a5f; color: white; padding: 1rem 2rem; }
    nav a { color: white; margin-right: 1rem; text-decoration: none; }
    nav a:hover { text-decoration: underline; }
    .layout { display: flex; gap: 2rem; padding: 2rem; }
    main { flex: 3; }
    aside { flex: 1; background: #f1f4f8; padding: 1rem; border-radius: 8px; }
    article { margin-bottom: 2rem; }
    table { border-collapse: collapse; width: 100%; }
    th, td { border: 1px solid #ccc; padding: 8px; text-align: left; }
    th { background: #e8eef5; }
    @media (max-width: 700px) { .layout { flex-direction: column; } }
  </style>
</head>
<body>

  <header>
    <h1>My Tech Blog</h1>
    <nav>
      <a href='#home'>Home</a>
      <a href='#articles'>Articles</a>
      <a href='#contact'>Contact</a>
    </nav>
  </header>

  <div class='layout'>
    <main id='home'>
      <section id='articles'>
        <h2>Latest Articles</h2>

        <article>
          <h3>Understanding REST APIs</h3>
          <p>Published on <time datetime='2026-11-10'>10 November 2026</time></p>
          <p>REST uses <strong>HTTP methods</strong> to manage
<em>resources</em>.</p>
          <figure>
```

```
            <img src='https://via.placeholder.com/400x150' alt='REST diagram
placeholder'>
            <figcaption>A client talking to a REST server</figcaption>
          </figure>
          <details>
            <summary>Show HTTP methods</summary>
            <ul>
              <li>GET</li><li>POST</li><li>PUT</li><li>DELETE</li>
            </ul>
          </details>
        </article>

        <article>
          <h3>Status Codes</h3>
          <table>
            <thead><tr><th>Code</th><th>Meaning</th></tr></thead>
            <tbody>
              <tr><td>200</td><td>OK</td></tr>
              <tr><td>404</td><td>Not Found</td></tr>
            </tbody>
          </table>
        </article>
      </section>

      <section id='contact'>
        <h2>Contact</h2>
        <form action='#' method='post'>
          <p><label for='n'>Name: </label><input id='n' name='n' type='text'
required></p>
          <p><label for='e'>Email: </label><input id='e' name='e' type='email'
required></p>
          <p><label for='m'>Message: </label><br><textarea id='m' name='m' rows='4'
cols='40'></textarea></p>
          <button type='submit'>Send</button>
        </form>
      </section>
    </main>

    <aside>
      <h2>Quick Links</h2>
      <ul>
        <li><a href='https://developer.mozilla.org' target='_blank'
rel='noopener'>MDN Web Docs</a></li>
        <li><a href='https://html.spec.whatwg.org' target='_blank'
rel='noopener'>HTML Standard</a></li>
      </ul>
    </aside>
  </div>

  <footer>
    <p>&copy; 2026 My Tech Blog. All rights reserved.</p>
    <address>Contact: <a href='mailto:me@example.com'>me@example.com</a></address>
  </footer>

  <script>
```

```
    document.querySelector('form').addEventListener('submit', function (e) {
      e.preventDefault();
      alert('Form submitted (demo only)');
    });
  </script>
</body>
</html>
```