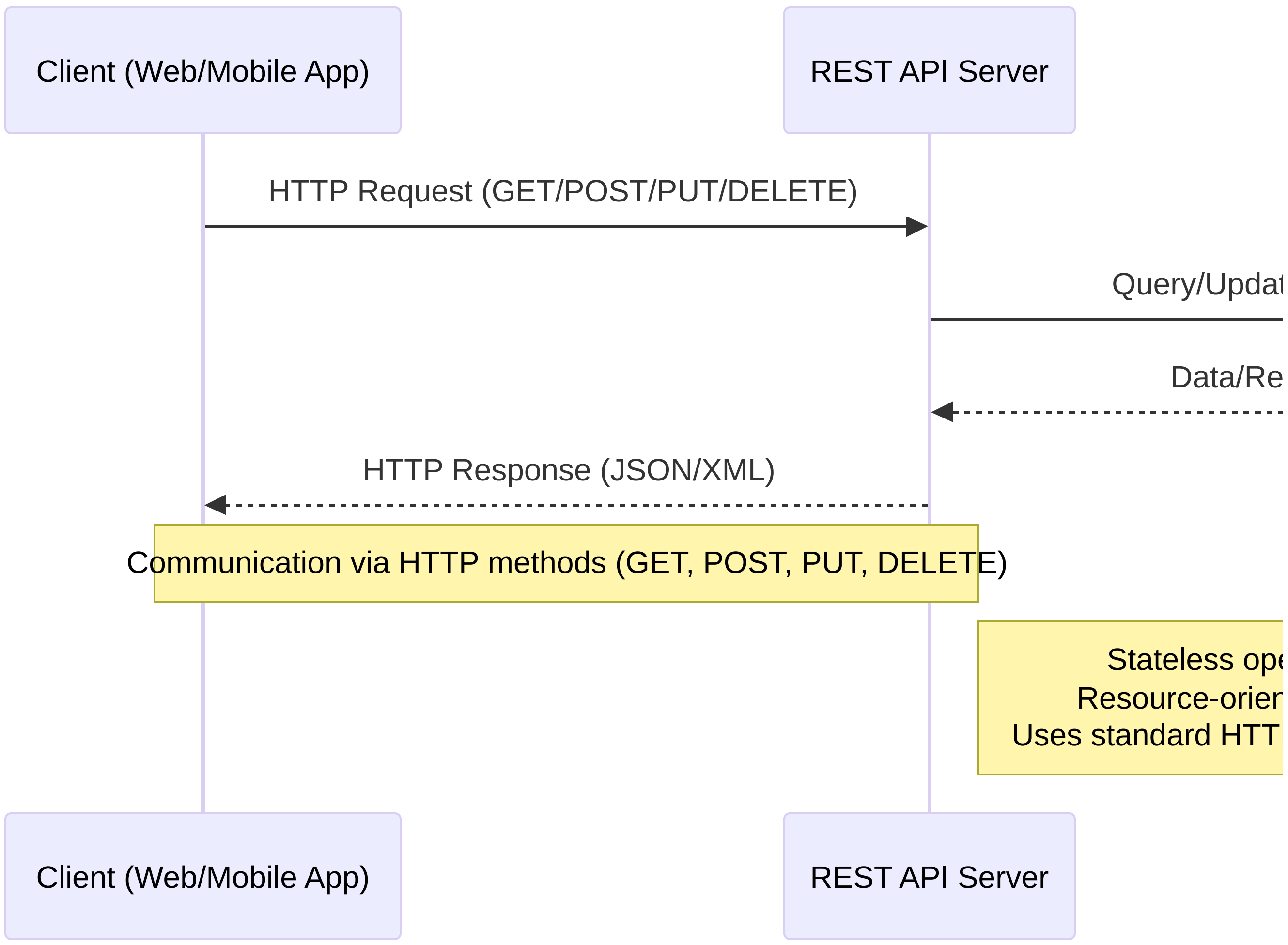

Explanation:

- **Client:** Can be a web browser, mobile app, or any service that wants to access data.
- **REST API Server:** Exposes endpoints (URLs) that the client can use. Handles requests, processes data, and returns responses.
- **Database:** Stores persistent data. The API server queries or updates it based on client requests.
- **Communication:** Uses HTTP methods (GET for retrieve, POST for create, PUT for update, DELETE for remove).

RESTful API Best Practices

1. Use Nouns for Resource URIs

- URIs should represent resources (e.g., `/users`, `/orders/123`).
- Avoid using verbs in URIs (e.g., `/getUser`).

2. Use HTTP Methods Correctly

- **GET**: Retrieve resources (should not modify data).
- **POST**: Create new resources.
- **PUT**: Update a resource entirely.
- **PATCH**: Update part of a resource.
- **DELETE**: Remove a resource.

3. Use Plural Nouns

- Prefer `/users` over `/user`.

4. Meaningful HTTP Status Codes

- `200 OK`: Request succeeded.
- `201 Created`: Resource successfully created.
- `204 No Content`: Successful request, no body returned.
- `400 Bad Request`: Request is invalid.
- `401 Unauthorized`: Authentication required.
- `403 Forbidden`: Not allowed.
- `404 Not Found`: Resource does not exist.
- `500 Internal Server Error`: Generic server error.

5. Consistent Data Formatting

- Use JSON as the default response format.
- Set `Content-Type: application/json` in headers.

6. Version Your API

- Add versioning to your API (e.g., `/v1/users`).
- Prevents breaking changes for clients.

7. Statelessness

- Each request should contain all required information (no session state stored on server).

8. Use Filtering, Sorting, and Pagination

- Allow clients to filter, sort, and paginate results:
  - `/users?role=admin`

- `/products?sort=price&order=asc`
- `/orders?page=2&limit=50`

9. Error Handling and Messages

- Return structured error responses:

```
{
  "error": {
    "code": 400,
    "message": "Invalid user ID"
  }
}
```

10. Secure Your API

- Use HTTPS.
- Implement authentication (e.g., JWT, OAuth).
- Validate and sanitize input.

11. Documentation

- Provide clear, up-to-date documentation (OpenAPI/Swagger is recommended).

12. HATEOAS (Optional for Advanced REST)

- Include links to related resources in responses for better discoverability.

Summary:
Following these best practices ensures your RESTful API is robust, user-friendly, secure, and maintainable.