CRUD Operations in Web Engineering

1 CRUD Operations in Web Development

1.1 What does CRUD stand for?

CRUD stands for **Create, Read, Update, and Delete** — these are the four basic operations of *persistent storage* used in web applications to manage data.

2 Explanation of Each Operation with Examples

Operation	**Description**	**Example (Web App)**
C - Create	Adds a **new** data entry to the system.	Register a new student by filling out a form.
R - Read	Retrieves and displays **existing** data.	View a list of all students or details of a specific student.
U - Update	Modifies **existing** data.	Edit a student's phone number or update their course.
D - Delete	**Removes** data from the system.	Delete a student's record after graduation.

3 Function-Centric vs Resource-Centric Web Services

Feature	**Function-Centric**	**Resource-Centric (REST)**
Design Focus	Actions / Functions	Real-world entities / Resources
Interaction Style	Custom function calls	Standard HTTP methods (GET, POST, PUT, DELETE)

Data Access	Based on operations	Based on URIs (Uniform Resource Identifiers)
Example Operation	`getPlaylist(userId)`	`GET /playlists/324`
Extensibility	Harder to add new functions	Easier to add new resource types
Standardization	Low – each function is unique	High – uses standard web methods
Protocol Support	Often uses SOAP, requires WSDL	Uses HTTP, typically with JSON
Learning Curve	Higher (due to custom interfaces)	Lower (uniform and intuitive)
Scalability	Difficult to scale due to tight coupling	Easy to scale and cache due to statelessness
Best Use Cases	Enterprise-level apps with complex logic	Web and mobile apps needing fast integration

4 How HTTP Methods Relate to CRUD

HTTP Method	**CRUD Operation**	**Description**
POST	Create	Submits data to the server to create a new resource.
GET	Read	Requests data from the server without changing it.
PUT / PATCH	Update	Sends updated data to the server to replace or modify an existing resource.
DELETE	Delete	Sends a request to the server to remove a resource.

5 CRUD in a Student Management System

Let's say we have a **Student Management System** for a university web application:

1. **Create (POST):**
   A form lets the admin *add a new student*. An HTTP POST request sends student info (name, ID, department) to the server, which stores it in the database.

2. **Read (GET):**
   The system displays a *list of all students* or searches for one by ID. An HTTP GET

request fetches student data from the database.

3. **Update (PUT or PATCH):**
   Admin wants to *update a student's email*. A form captures the new email, and an HTTP PUT request sends the updated info. The server modifies the existing record.

4. **Delete (DELETE):**
   A student is *removed from the system* (e.g., dropped out). Admin clicks "Delete," sending an HTTP DELETE request to remove the record from the database.

6 Summary

- CRUD stands for **Create, Read, Update, Delete** — the four core functions of any data-driven web application.
- CRUD maps to **HTTP methods**: `POST` → `Create`, `GET` → `Read`, `PUT` → `Update`, `DELETE` → `Delete`.
- In real systems like a *Student Management System*, CRUD is used to manage student data through web forms and backend processing.

7 What is a REST API and why is it used?

REST stands for **Representational State Transfer**. A REST API (Application Programming Interface) is a set of rules that allows web applications to communicate over the internet using standard HTTP methods.

Key benefits:

- Uses standard HTTP methods: GET, POST, PUT, DELETE
- Lightweight and fast
- Easy to implement and scale
- Works well with JSON or XML over the web

In short, REST APIs connect frontend applications (like browsers) to backend services (like servers or databases) using a simple protocol.

8 What is a Resource in REST?

In REST architecture, everything is treated as a **resource**, which is any identifiable object with a unique URI (Uniform Resource Identifier).

Example: Student Management System

- `GET /students` → Get list of all students
- `GET /students/101` → Get student with ID 101
- `POST /students` → Add a new student
- `PUT /students/101` → Update student 101
- `DELETE /students/101` → Delete student 101

Resources are **nouns** (e.g., students, books) and HTTP methods represent actions.

9 How HTTP Requests Work in REST

Each HTTP method maps to a specific CRUD operation:

HTTP Method	Action	Description
GET	Read	Retrieve data from the server.
POST	Create	Add new data to the server.
PUT	Update	Replace existing data.
PATCH	Update	Partially update data.
DELETE	Delete	Remove data from the server.

REST is **stateless** — each request is independent and contains all the information needed.

10 Why REST is Developer-Friendly

- Simple and clear structure using URLs and HTTP verbs
- Stateless: No need for server memory or sessions
- Supports JSON — easy to use in web development
- Scalable and fast to implement
- Supported by tools like Postman, Swagger, and browser dev tools

11 REST vs SOAP – Comparison Table

Aspect	REST	SOAP
Protocol	Uses standard HTTP methods (GET, POST, PUT, DELETE)	Uses XML-based protocol over HTTP, SMTP, etc.
Complexity	Lightweight, simple, predictable structure	More complex, uses WS-* specifications and standards
Message Format	Usually JSON or XML	Strictly XML
API Contract	No strict contract (no WSDL/XSD required)	Requires WSDL and XSD files (strict contracts)
Client Integration	No auto-generated clients, flexible, easier to update without breaking clients	Supports auto-generated client code, but changes require redeployment
Security	Uses HTTPS/TLS and OAuth tokens; simpler security	Supports WS-Security with message-level encryption and advanced features
State	Stateless; easy to cache GET requests for scalability	Stateful or stateless; caching is limited
Scalability	Highly scalable due to statelessness and HTTP caching	Less scalable due to complexity and stateful nature
Error Handling	Uses standard HTTP status codes	Uses detailed XML-based error messages
Tooling & Standards	Minimal required tooling, can use any web stack	Requires more tooling and standards compliance
Best Use Case	Web/mobile apps, startups, lightweight integrations	Enterprise systems needing strict contracts, advanced security, transactions

11.1 Summary

- REST API uses HTTP to manage resources (like `/students`)
- REST is lightweight, scalable, and suitable for modern web apps
- SOAP is better suited for enterprise-level services with advanced security and strict contracts

- An **API contract** is a formal agreement between the API provider and the API consumer (client)
• **WSDL:** Web Services Description Language and **XSD:** XML Schema Definition
• **Client integration** means connecting your application (the client) to an external web service or API so it can send requests and receive responses

12 REST API: Request Types, Backend Handling, and Status Codes

12.1 What Kind of Requests Can Be Sent to a REST API?

REST APIs primarily use five HTTP methods to interact with resources:

Method	Purpose	Example	Description
GET	Read	`GET /students/101`	Retrieve details of student with ID 101.
POST	Create	`POST /students`	Add a new student (data is sent in the request body).
PUT	Update (Full)	`PUT /students/101`	Replace entire record of student 101 with new data.
PATCH	Update (Partial)	`PATCH /students/101`	Modify specific fields (e.g., phone number) only.
DELETE	Delete	`DELETE /students/101`	Remove student record with ID 101.

Each HTTP method clearly communicates what action is expected on a given resource.

12.2 How Does the Backend Handle HTTP Methods?

The backend uses **routing** to identify the HTTP method and call the appropriate function to handle the request.

Example logic (Node.js Express-style syntax):

```
// GET - Read data
app.get('/students/:id', (req, res) => {
  // Fetch student by ID
});

// POST - Add new data
app.post('/students', (req, res) => {
  // Insert new student data
});

// PUT - Replace full record
app.put('/students/:id', (req, res) => {
  // Replace full student object
});
```

```
// PATCH - Update part of a record
app.patch('/students/:id', (req, res) => {
  // Update only specific fields
});

// DELETE - Remove record
app.delete('/students/:id', (req, res) => {
  // Delete student from database
});
```

Each route runs method-specific logic, often connected to a database or storage system.

12.3 Importance of HTTP Status Codes

HTTP status codes indicate the result of an API request, helping clients understand whether a request succeeded or failed.

Code	Meaning	When to Use
200 OK	Success	Used when resource is fetched or action completed successfully.
201 Created	New Resource Created	Used after a successful POST request.
204 No Content	Success with no response body	Used with DELETE or empty success responses.
400 Bad Request	Invalid Request	Input data is incorrect or missing.
401 Unauthorized	Authentication Required	Client must provide valid credentials.
404 Not Found	Resource Not Found	Resource does not exist in database.
500 Internal Server Error	Server Failure	Unexpected issue during request processing.

12.3.1 Why Status Codes Matter

- Help clients understand whether the request was successful.
- Improve error handling and debugging.
- Maintain standardized communication between client and server.

12.3.2 Summary

- REST supports **GET, POST, PUT, PATCH, DELETE** requests for managing resources.

- Backend uses routing logic to handle each method distinctly.
- **HTTP status codes** are essential for clarity, debugging, and smooth API usage.

13 Backend Frameworks, APIs, and Frontend-Backend Decoupling

13.1 How Do Web Frameworks (like ExpressJS, Django, Laravel) Help in Faster Backend Development?

- **Pre-built tools & libraries:** Frameworks come with ready-to-use components (routing, database handling, authentication) that save time.
- **Standardized structure:** They provide a clear project architecture, reducing confusion and speeding up coding.
- **Security & best practices:** Frameworks handle common security issues (e.g., SQL injection, CSRF) out-of-the-box.
- **Community & plugins:** Large ecosystems mean lots of plugins and documentation to quickly add features.
- **Less boilerplate code:** Automate repetitive tasks so developers focus on business logic.

Result: Developers build, test, and deploy backend services much faster than coding from scratch.

13.2 Why Are APIs Called a "Fast Approach" for Developing Modern Applications?

- **Reusability:** APIs allow reuse of existing backend logic and services across multiple clients (web, mobile, IoT).
- **Parallel development:** Frontend and backend teams can work independently by agreeing on API contracts.
- **Faster integration:** Easily connect with third-party services or microservices without reinventing the wheel.
- **Scalability:** APIs can serve many clients simultaneously with minimal changes.

- **Rapid prototyping:** Developers can quickly expose data/functionality via APIs to build MVPs or prototypes.

13.3 How Do REST APIs Help Connect Frontend and Backend in a Decoupled Way?

- **Separation of concerns:** Frontend only calls API endpoints to get or send data, without knowing backend implementation.
- **Flexibility:** Frontend can be built with any technology (React, Angular, Vue) independently of backend (Node.js, Django, etc.).
- **Stateless communication:** Each API request contains all needed info; server does not keep session, enabling scalability.
- **Ease of maintenance:** Backend and frontend can evolve independently as long as API contracts remain stable.
- **Cross-platform:** Same REST API can serve web apps, mobile apps, and other clients consistently.

13.4 Summary

- Frameworks speed up backend development with tools, security, and structure.
- APIs enable rapid app development by allowing reuse and parallel work.
- REST APIs decouple frontend and backend, improving flexibility and maintainability.

14 Application and Resource Modeling in REST API Design

14.1 Why is it important to model real-world concepts as resources in REST API?

- **Clarity & Intuition:** Resources represent real-world entities (like students, results), making APIs easier to understand and use.
- **Uniformity:** Treating everything as a resource allows uniform access via URIs (e.g., `/students/123`).
- **Reusability:** Resources can be independently managed, reused, or linked to other resources.

- **Scalability:** Resource-based design naturally supports scaling by focusing on discrete entities.
- **Flexibility:** Easier to extend and maintain as new resource types can be added without changing the API style.

14.2 How can a 'student' be treated as a resource in a result processing system?

- A **student** represents a single, identifiable entity with attributes like ID, name, class, and enrollment details.
- It has a **unique URI** such as `/students/{studentId}`.
- Actions on a student (viewing details, updating info, deleting record) are performed through HTTP methods on this resource URI.
- A student resource can be linked to other resources like **results** (e.g., `/students/{studentId}/resul` representing the exam scores.

14.3 Designing REST API endpoints for a result management system

Resource	HTTP Method	Endpoint	Description
Students	GET	`/students`	Retrieve all students.
	GET	`/students/{id}`	Retrieve a specific student by ID.
	POST	`/students`	Add a new student.
	PUT	`/students/{id}`	Update all info of a student.
	PATCH	`/students/{id}`	Update part of a student's data.
	DELETE	`/students/{id}`	Delete a student.
Results	GET	`/results`	Get all results.
	GET	`/results/{resultId}`	Get specific result record.
	GET	`/students/{id}/results`	Get all results for a specific student.
	POST	`/students/{id}/results`	Add new result for a student.
	PUT	`/results/{resultId}`	Update a result record fully.
	DELETE	`/results/{resultId}`	Delete a result record.

Summary

- Modeling real-world entities as **resources** creates a simple, intuitive API design.
- A **student** resource is uniquely identifiable and interacts via HTTP methods.
- Designing REST endpoints around resources (`/students`, `/results`) helps organize data access clearly and logically.

15 Example: Student Resource CRUD Operations

1. Create Student

Request:

```
POST /students
Content-Type: application/json

{
  "name": "Alice Rahman",
  "age": 20,
  "department": "Computer Science"
}
```

Response:

```
201 Created
Content-Type: application/json

{
  "id": 101,
  "name": "Alice Rahman",
  "age": 20,
  "department": "Computer Science"
}
```

2. Read/Get All Students

Request:

```
GET /students
```

Response:

```
200 OK
Content-Type: application/json

[
  {
    "id": 101,
    "name": "Alice Rahman",
    "age": 20,
    "department": "Computer Science"
  },
  {
    "id": 102,
    "name": "Bob Karim",
    "age": 22,
    "department": "Electrical Engineering"
  }
]
```

3. Read/Get Single Student

Request:

```
GET /students/101
```

Response:

```
200 OK
Content-Type: application/json

{
  "id": 101,
  "name": "Alice Rahman",
  "age": 20,
  "department": "Computer Science"
}
```

4. Update Student (Full Update)

Request:

```
PUT /students/101
Content-Type: application/json

{
  "name": "Alice Rahman",
  "age": 21,
  "department": "Computer Science"
}
```

Response:

```
200 OK
Content-Type: application/json

{
  "id": 101,
  "name": "Alice Rahman",
  "age": 21,
  "department": "Computer Science"
}
```

5. Update Student (Partial Update)

Request:

```
PATCH /students/101
Content-Type: application/json

{
  "age": 22
}
```

Response:

```
200 OK
Content-Type: application/json

{
  "id": 101,
  "name": "Alice Rahman",
```

```
  "age": 22,
  "department": "Computer Science"
}
```

6. Delete Student

Request:

```
DELETE /students/101
```

Response:

```
204 No Content
```

16 Example: Result Resource Operations

1. Add Result for a Student

Request:

```
POST /students/101/results
Content-Type: application/json

{
  "subject": "Mathematics",
  "score": 88
}
```

Response:

```
201 Created
Content-Type: application/json

{
  "resultId": 501,
  "studentId": 101,
  "subject": "Mathematics",
  "score": 88
}
```

2. Get All Results for a Student

Request:

```
GET /students/101/results
```

Response:

```
200 OK
Content-Type: application/json

[
  {
    "resultId": 501,
    "subject": "Mathematics",
    "score": 88
  },
  {
    "resultId": 502,
    "subject": "Physics",
    "score": 92
  }
]
```

3. Get Single Result

Request:

```
GET /results/501
```

Response:

```
200 OK
Content-Type: application/json

{
  "resultId": 501,
  "studentId": 101,
  "subject": "Mathematics",
  "score": 88
}
```

4. Update Result

Request:

```
PUT /results/501
Content-Type: application/json

{
  "subject": "Mathematics",
  "score": 90
}
```

Response:

```
200 OK
Content-Type: application/json

{
  "resultId": 501,
  "studentId": 101,
  "subject": "Mathematics",
  "score": 90
}
```

5. Delete Result

Request:

```
DELETE /results/501
```

Response:

```
204 No Content
```