Advice for Demo

When presenting:

1. Start with architecture
2. Show feature file
3. Show step definition
4. Show API class
5. Show utility engine
6. Run test
7. Show report

Q1: Why did you choose layered architecture?

Expected Answer Direction:

- Separation of concerns
- Maintainability
- Reusability
- Cleaner scaling

Q2: Why did you separate API classes (GET, POST, PUT, DELETE)?

They may ask:

Why not write everything inside step definitions?

You should say:

- Keeps step definitions clean
- Business logic abstraction
- Reusable across scenarios
- Follows SRP (Single Responsibility Principle)

Q3: Why centralize HTTP logic in ApiMethods?

They want to check if you understand abstraction.

Correct reasoning:

- Avoid duplication
- Easy to switch HTTP client

- Easy to add logging/retry
- Easier debugging

Q4: What design principles does your framework follow?

You should mention:

- SOLID principles
- DRY
- Separation of concerns
- Abstraction
- Encapsulation

Q5: How would you scale this framework for 200 APIs?

Expected thinking:

- Group APIs by domain
- Add service folders
- Add environment configs
- Add base service layer
- Add reusable request wrapper

🔵 2️⃣ Utility Layer Deep Questions

Q6: Why use destructured object in request() instead of parameters?

They are checking if you understand:

```
request({ method, url, endpoint })
```

Answer:

- Improves readability
- Allows optional parameters
- Prevents argument order mistakes

Q7: Why use deepClone()?

To prevent:

- Mutation of base payload
- Side effects across tests

Q8: Why use AJV schema validation?

To validate:

- Response contract
- Data structure
- Prevent breaking changes

Q9: What happens if schema changes?

Answer:

- Tests fail immediately
- Detects contract break early
- Helps in API version control

Q10: How is token handled and why cookie?

Because:

- Restful Booker expects token as cookie
- Centralized authentication logic

🟢 3️⃣ BDD & Cucumber Questions

Q11: What is the role of Cucumber in this framework?

Answer:

- Enables BDD
- Makes tests readable
- Bridges business and automation

Q12: What is the World object?

Very common question.

Correct explanation:

- Scenario-level shared context
- Each scenario gets new instance
- Used to share bookingId

Q13: Why use DataTable in feature file?

Because:

- Dynamic input
- Cleaner test readability
- Supports multiple combinations

Q14: What happens internally when Cucumber runs?

Expected explanation:

- Reads feature file
- Matches step definitions
- Executes steps sequentially
- Maintains isolated World context

🟡 4 API Testing Concept Questions

Q15: Difference between PUT and PATCH?

PUT → Replaces entire resource
PATCH → Partial update

Q16: Why validate status code separately?

Because:

- Status code indicates protocol-level success
- Schema validates data-level correctness

Q17: What is idempotent method?

GET, PUT, DELETE → Idempotent
POST → Not idempotent

Q18: How do you handle negative testing?

Possible answers:

- Pass invalid payload
- Validate 400/401/404 responses
- Add schema for error responses

🔴 5 Scalability & Enterprise Questions

Q19: How would you add environment support (dev/qa/prod)?

Answer:

- Add .env file
- Add environment config file
- Use process.env

Q20: How would you integrate this into CI/CD?

Answer:

- Use GitHub Actions
- Run npm run spec
- Generate reports
- Archive reports
- Fail pipeline on test failure

Q21: How to run tests in parallel?

Possible solutions:

- Use cucumber parallel flag
- Split feature files
- Use worker threads

Q22: How to rerun failed tests?

Answer:

- Generate failed.json
- Use tag rerun
- Use retry mechanism

🔴 6 Demo-Time Practical Questions

During demo they may ask:

Q23: What happens if bookingId is not stored?

Answer:

- Defensive check throws error
- Prevents invalid API call

Q24: What happens if server returns 500?

Answer:

- validateStatus() throws error
- Test fails immediately

Q25: How do you debug failures?

Answer:

- Error message shows:
  - Method
  - Endpoint
  - Expected
  - Actual
  - Response body

● 7 Scenario-Based Questions

Q26: If token expires after 10 minutes?

Answer:

- Implement token caching
- Add auto-refresh logic
- Store expiry timestamp

Q27: If API response time increases?

Answer:

- Add timeout
- Add performance threshold check

Q28: How to test performance in this framework?

Answer:

- Add response time validation
- Integrate with k6 or JMeter

● 8 Advanced / Senior-Level Questions

These check deep understanding.

Q29: How is async/await working internally?

Expected:

- Returns promise
- Uses event loop
- Non-blocking I/O

Q30: How would you convert this to microservice testing framework?

Answer:

- Create service-wise folders
- Add service-level base class
- Add shared auth handler
- Add API client abstraction

Q31: If company moves from Supertest to Axios?

Answer:

- Only change ApiMethods.js
- API classes remain unchanged
- That’s the benefit of abstraction

Q32: How to add retry logic?

Add inside request():

```
for (let i = 0; i < 3; i++) {
   try { ... } catch {}
}
```

Q33: What are limitations of this framework?

Good to mention honestly:

- No logging yet
- No retry mechanism
- No environment config
- No parallel execution
- No centralized error handling

This shows maturity.

What is the World Object?

Very common Cucumber question.

◇ Definition

World object is:

A scenario-level shared context provided by Cucumber.

Each scenario gets a fresh instance.

◇ Why It's Needed?

In your scenario:

1. Create booking
2. Retrieve booking
3. Update booking
4. Delete booking

Step 1 generates:

```
this.world.bookingId = response.body.bookingid;
```

Step 2 uses it:

```
const bookingId = this.world?.bookingId;
```

Without World object:
You cannot share bookingId across steps.

◇ Important Property

Each scenario has its own isolated World.

Example:

Scenario 1 → bookingId = 123
Scenario 2 → bookingId = 456

They do NOT conflict.

◇ What Happens Internally?

Cucumber does:

```
For each scenario:
   create new World instance
   execute steps
   destroy instance
```

This prevents state leakage.

◇ Interview-Ready Explanation

If asked:

What is World object?

You say:

The World object is a scenario-scoped context provided by Cucumber. It allows sharing state between steps within the same scenario. Each scenario gets its own isolated World instance to prevent cross-test contamination.

---

3 What is an Idempotent Method?

This is a pure API concept.

◇ Definition

An HTTP method is idempotent if:

Making the same request multiple times produces the same result.

◇ In Simple Words

Repeat the request → No additional side effects.

◇ Idempotent Methods

Method	Idempotent?
GET	Yes
PUT	Yes

Method	Idempotent?
DELETE	Yes
POST	No

◇ Example Using Your Framework

☑ GET (Idempotent)

```
GET /booking/1
GET /booking/1
GET /booking/1
```

Returns same booking every time.

No change to server.

☑ PUT (Idempotent)

```
PUT /booking/1
{ "firstname": "Sai" }
```

If you run it 10 times:

Final state is still:

```
firstname = Sai
```

No extra changes.

☑ DELETE (Idempotent)

First time:

```
DELETE /booking/1
```

Deletes booking.

Second time:

```
DELETE /booking/1
```

It might return 404, but resource is still deleted.

State remains same.

So idempotent.

❌ POST (Not Idempotent)

```
POST /booking
```

If you run it 5 times:

You create 5 bookings.

Each request changes system state.

So not idempotent.

◇ Why Interviewers Ask This?

To test:

- Your REST knowledge
- Your API understanding
- Your backend fundamentals

◇ Interview-Ready Answer

An idempotent method is one where multiple identical requests produce the same final state on the server. GET, PUT, and DELETE are idempotent because repeating them doesn't change the result after the first successful execution. POST is not idempotent because each call creates a new resource.

1 "Why didn't you directly use Axios instead of Supertest?"

😈 Why this is tricky:

They want to test whether you understand:

- Tool selection
- Framework purpose
- Testing vs API client usage

☑ Strong Answer:

I chose Supertest because it's optimized for API testing scenarios and integrates well with test assertions. It simplifies request building and response validation. However, since HTTP logic is abstracted inside ApiMethods, I can easily switch to Axios without modifying API classes. That flexibility was intentional in my design.

🚩 Weak Answer:

I just saw Supertest in a tutorial.

🔥 2️⃣ “What happens if two scenarios run in parallel?”

😈 Why tricky:

Tests your understanding of World object isolation.

☑ Strong Answer:

Each Cucumber scenario gets its own World instance, so bookingId won’t collide. However, if the external API shares state globally, conflicts may happen. For true parallel safety, I would ensure unique test data generation and avoid shared global variables.

🔥 3️⃣ “Is your Utility class violating Single Responsibility Principle?”

😈 Why tricky:

Because ApiMethods handles:

- HTTP
- Validation
- Token
- Payload

They want to see if you understand SRP deeply.

☑ Strong Answer:

Currently ApiMethods acts as a centralized HTTP engine. In enterprise-level systems, I would split it into:

- HttpClient
- Validator
- AuthService
  But for this scale, combining them improves maintainability without overengineering.

Shows maturity.

🔥 4️⃣ "Why store bookingId in World instead of returning it?"

😈 Why tricky:

They're testing state management understanding.

☑️ Strong Answer:

In BDD, steps are independent units. Returning bookingId wouldn't automatically share it across steps. World acts as scenario-scoped shared memory, enabling stateful workflow execution while maintaining isolation between scenarios.

🔥 5️⃣ "What if the API response schema changes but status is still 200?"

😈 Why tricky:

Many candidates forget schema validation.

☑️ Strong Answer:

That's exactly why I integrated AJV schema validation. Even if status is 200, if response structure changes, validateSchema() will fail the test, preventing silent contract breakage.

🔥 6️⃣ “Is DELETE really idempotent if second call gives 404?”

😈 Why tricky:

This confuses many candidates.

☑️ Strong Answer:

Yes. Idempotency is about the final state, not the response code. After the first DELETE, the resource no longer exists. Subsequent DELETE calls don’t change the system state further, even if they return 404. Therefore, DELETE is idempotent.

🔥 7️⃣ “Your framework handles token generation per request. Isn’t that inefficient?”

😈 Why tricky:

Tests optimization thinking.

☑️ Strong Answer:

Yes, generating token per request is not optimal. In production, I would implement token caching with expiry handling. For simplicity and isolation in this demo, I regenerate token per secured request.

🔥 8️⃣ “If tomorrow we move to microservices, what changes?”

😈 Why tricky:

Tests architecture scalability.

☑️ Strong Answer:

I would group APIs by service domain and create service-specific base classes. Auth handling might become centralized per service. The utility layer remains unchanged because HTTP abstraction already exists.

🔥 9️⃣ “What are limitations of your framework?”

😈 Why tricky:

Honesty + awareness test.

☑️ Strong Answer:

Current limitations:

- No retry mechanism
- No logging framework
- No environment switching
- No parallel execution config
- No test data randomization strategy

These can be added without structural redesign due to modular architecture.

🔥 🔟 “What happens if request() throws an exception?”

😈 Why tricky:

Tests async error flow understanding.

☑️ Strong Answer:

The exception bubbles up to API class, then to step definition. Since we use async/await, unhandled promise rejections fail the scenario immediately. I could enhance it with centralized error handling middleware.

🔥 1️⃣1️⃣ “Is your framework truly reusable?”

😈 Why tricky:

They want architectural justification.

☑ Strong Answer:

Yes, because business logic is separated from transport logic. I can plug this into another REST API by simply changing endpoints and payloads without modifying utility engine.

🔥 12️⃣ “Why use class-based approach instead of functional?”

😈 Why tricky:

Tests design reasoning.

☑ Strong Answer:

I chose classes for encapsulation and state handling (like world object integration). It also makes it easier to extend behavior via inheritance if needed.

🔥 13️⃣ “If API becomes GraphQL tomorrow?”

😈 Why tricky:

Tests adaptability.

☑ Strong Answer:

Since transport logic is abstracted in request(), I would modify request() to support GraphQL queries. API layer and step definitions would remain largely unchanged.

🔥 14️⃣ “What if Supertest is deprecated?”

😈 Why tricky:

Tests abstraction understanding.

☑ Strong Answer:

Only ApiMethods.js would change. That's the benefit of abstraction. API classes and test logic would remain untouched.

🔥 15️⃣ "Can your framework support contract testing?"

😈 Why tricky:

Tests advanced knowledge.

☑ Strong Answer:

Currently it supports schema validation, which is partial contract testing. For full contract testing, I would integrate OpenAPI validation or Pact.

🧠 Ultimate Trick Question

❓ "Is your framework stateless?"

😈 Why tricky:

Because World object stores state.

☑ Strong Answer:

The framework maintains scenario-level state using World, but it does not maintain global shared state across scenarios. Therefore, it is stateless at execution level but stateful within scenario context.

Why did you create a Utility (ApiMethods) class?

Question:

Why didn’t you just write the HTTP request code inside POST, GET, PUT, DELETE files?

Simple Answer:

I created the Utility class to avoid repeating the same HTTP logic everywhere.
If I wrote the request logic in every API file, it would duplicate code.

Now, all HTTP handling is in one place.
If I need to change headers, add logging, or switch libraries, I change it in one file only.

---

2 What is the World object?

Question:

What is World in Cucumber and why are you using it?

Simple Answer:

The World object is like memory for one scenario.

When I create a booking, I get a bookingId.
I store that bookingId inside World.

Then in the next step (like GET or PUT), I can use the same bookingId.

Each scenario gets its own World object, so scenarios don’t interfere with each other.

---

3 What happens if two scenarios run at the same time?

Simple Answer:

Each scenario has its own World object, so bookingId will not clash.

However, if both scenarios use the same data on the server, then conflicts can happen.
To avoid that, I would generate unique test data.

---

4 Why is POST not idempotent?

Question:

What is idempotent method?

Simple Answer:

An idempotent method means if you send the same request multiple times, the result stays the same.

GET is idempotent because it just fetches data.
PUT is idempotent because updating the same value again does not change anything.
DELETE is idempotent because deleting again doesn't change the state.

POST is not idempotent because every time you send it, it creates a new resource.

---

5 Why are you generating token every time in PUT and DELETE?

Simple Answer:

For simplicity and isolation, I generate token before secured requests.

In a real project, I would cache the token and reuse it until it expires.

---

6 Why use schema validation?

Simple Answer:

Even if API returns status 200, the response structure might change.

Schema validation ensures the response format is correct.
If something changes in the structure, the test fails immediately.

---

7 What design principles does your framework follow?

Simple Answer:

My framework follows:

- Single responsibility (each class has one job)
- Don’t repeat yourself (HTTP logic is centralized)
- Separation of concerns (feature, step, API, utility are separated)
- Abstraction (step definitions don’t know about HTTP details)
- Encapsulation (token handling is hidden inside utility)

---

8 Why didn’t you write everything in step definitions?

Simple Answer:

If I write everything in step definitions, the file becomes messy and hard to maintain.

By separating API logic into classes, the framework is cleaner and scalable.

---

9 What happens if API returns 500?

Simple Answer:

The validateStatus() method will detect that the response status is not expected and throw an error.
The test will fail immediately.

🔟 What if Supertest changes tomorrow?

Simple Answer:

Only the Utility file needs to change.
All API classes and step definitions will remain the same.

That’s why I centralized HTTP logic.

1️⃣1️⃣ What are limitations of your framework?

Simple Answer:

Currently:

- No retry logic
- No logging system
- No environment switching
- No parallel execution setup

But the structure allows easy addition of these features.

1️⃣2️⃣ Why use async/await?

Simple Answer:

API calls take time.
async/await makes the code wait for the response properly and keeps it readable.

👨‍💼 Interviewer:

Can you briefly explain your API automation framework?

👨‍💻 You:

Yes. I built a BDD-based API automation framework using Cucumber and Supertest.
The framework has a clear layered structure — feature files for business steps, step definitions for glue logic, API classes for business operations, and a Utility class that handles HTTP requests.
All HTTP logic is centralized, which makes the framework clean and scalable.

👨‍💼 Interviewer:

Why did you create a separate Utility class?

👨‍💻 You:

To avoid repeating HTTP request logic in every API file.
If I need to change headers, add logging, or switch the HTTP library, I only modify one file.
This keeps the framework maintainable and follows the DRY principle.

👨‍💼 Interviewer:

What is the World object in your framework?

👨‍💻 You:

The World object is scenario-level memory provided by Cucumber.
When I create a booking, I store the bookingId in the World object.
Other steps in the same scenario use that bookingId.
Each scenario gets its own World instance, so there's no data conflict.

👨‍💼 Interviewer:

What design principles does your framework follow?

👨‍💻 You:

It follows:

- Single Responsibility Principle (each class has one job)
- DRY (no duplicate HTTP code)
- Separation of concerns (feature, API, utility layers separated)
- Abstraction (step definitions don't know HTTP details)
- Encapsulation (token logic is hidden inside Utility)

👨‍💼 Interviewer:

What happens if the API response structure changes but status is still 200?

👨‍💻 You:

The test will fail because I added schema validation using AJV.
Even if status is 200, if the response format changes, validation will catch it.

👨‍💼 Interviewer:

Why is POST not idempotent?

👨‍💻 You:

Because every POST request creates a new resource.
If I send the same POST request multiple times, it creates multiple bookings.
So the result changes each time — that's why it's not idempotent.

👨‍💼 Interviewer:

How would you improve this framework?

👨‍💻 You:

I would add:

- Environment configuration (dev/qa/prod)
- Logging framework
- Retry mechanism
- Token caching
- Parallel execution setup

The structure already supports adding these easily.

👨‍💼 Interviewer:

If tomorrow the company moves from Supertest to Axios, what will you do?

👨‍💻 You:

I would only modify the Utility class.
Because HTTP logic is centralized, API classes and step definitions would not change.

👨‍💼 Interviewer:

What are the limitations of your framework?

👨‍💻 You:

Currently:

- No retry logic

- No centralized logging
- No environment switching
- Token generated per request (not cached)

What happens if two API calls depend on each other and the first one fails?

👨‍💻 Strong Answer:

If the first API call fails, the validateStatus method throws an error.
Because we are using async/await, the error bubbles up and the scenario stops immediately.

This prevents the second API call from running with invalid data.

👩‍💼 Interviewer:

How would you handle flaky APIs?

👨‍💻 Strong Answer:

If the API is unstable, I would implement:

- Retry logic inside the request() method
- Configurable retry count
- Logging of retry attempts

But retries should be used carefully — we shouldn’t hide real bugs.

👩‍💼 Interviewer:

How would you make this framework support multiple environments like dev, QA, and prod?

👨‍💻 Strong Answer:

I would:

- Create separate environment config files
- Store base URLs in .env file
- Use process.env to switch environments
- Pass environment variable during execution

For example:

```
cross-env ENV=qa npm run spec
```

Then endpoints would load based on environment.

👨‍💼 Interviewer:

How does async/await work internally?

👨‍💻 Strong Answer:

async functions return a Promise.
When we use await, JavaScript pauses execution of that function until the Promise resolves, but it does not block the entire thread.

Node.js uses an event loop, so other operations can continue while waiting for the API response.

👨‍💼 Interviewer:

What if your token expires in the middle of execution?

👨‍💻 Strong Answer:

Currently, token is generated before secured calls.

In a real-world project, I would:

- Cache token
- Store expiry time
- Refresh token automatically when expired

That would improve performance and reliability.

👨‍💼 Interviewer:

Is your framework stateless?

👨‍💻 Strong Answer:

It is stateless globally, but stateful within a scenario.

Each scenario maintains its own state using the World object.
There is no shared global state between scenarios.

👨‍💼 Interviewer:

How would you test negative scenarios?

👨‍💻 Strong Answer:

I would:

- Send invalid payloads
- Send unauthorized requests
- Validate 400/401/404 responses
- Add schema validation for error responses

Negative testing ensures API behaves correctly under failure conditions.

👨‍💼 Interviewer:

What if the API response time increases significantly?

👨‍💻 Strong Answer:

I would add response time validation inside request() like:

- Measure response time
- Fail test if it exceeds threshold

For performance testing at scale, I would use tools like k6 or JMeter.

👨‍💼 Interviewer:

What is the biggest weakness in your current framework?

👨‍💻 Strong Answer:

Currently:

- No centralized logging system
- No retry mechanism
- No token caching

- No parallel execution setup

But the structure allows easy improvement without major redesign.

🔥 Now a Slightly Tricky One

👨‍💼 Interviewer:

If I remove the Utility class and write everything inside step definitions, what problem would you face?

👨‍💻 Strong Answer:

The step definition file would become very large and messy.
HTTP logic would be duplicated across steps.
Maintaining or switching HTTP libraries would be difficult.
It would violate separation of concerns and make the framework harder to scale.

🎯 Final Impression Question

👨‍💼 Interviewer:

Why should we consider your framework production-ready?

👨‍💻 Strong Answer:

Because it is modular, cleanly structured, and easy to extend.
HTTP logic is centralized, business logic is separated, and schema validation ensures API contract stability.
It’s designed to grow without major restructuring.