

Module 4 - Backend Server

Developing RESTful APIs with Spring Boot

Recap: Spring Foundations

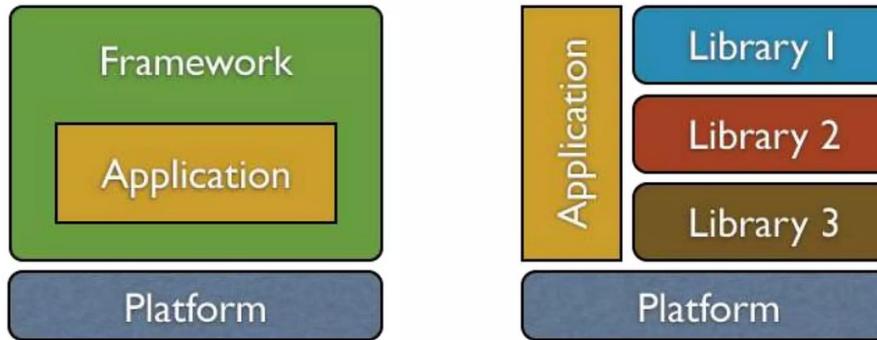
Section Overview

- Today we make Spring feel non-magical by mapping it to one concrete mental model.
- We will use TinyContainer as the X-ray view of what Spring is doing under the hood.
- Goal: you should be able to
 - explain where beans live
 - how they are created, and
 - when user code runs

TinyContainer reference: com.guvi.module4.student.main.AppMain (the main that starts everything)

Framework vs Library

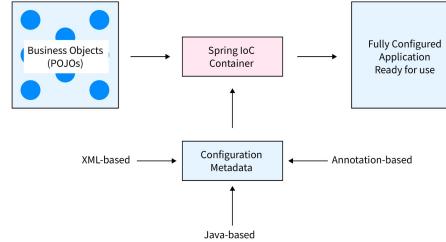
- A library is something your code calls to do work.
- A framework is something that calls your code at the right time, in the right order.
- Spring is a framework because it controls startup, object creation, wiring, and lifecycle.



TinyContainer reference: `TinyContainer.start()` runs startup tasks, which is the framework calling your code.

Inversion of Control (IoC)

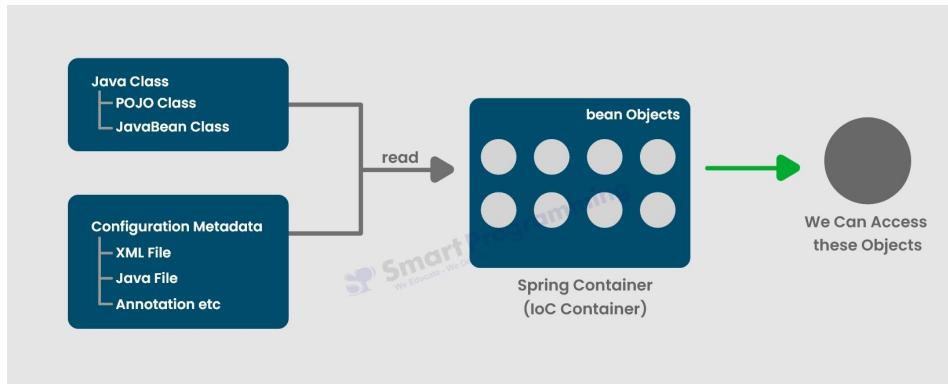
- Inversion of Control means the framework owns the control flow of the program.
- Your code stops being main drives everything and becomes declare components, framework runs them.
- IoC is the reason Spring can manage startup order, dependency wiring, and lifecycle consistently.



TinyContainer reference: IoC begins when AppMain stops using *new keyword for object creation* and instead uses registration + container.start().

What is a Container?

- A container is an object that creates, stores, and returns application objects for you.
- It becomes the central place that resolves dependencies between objects.
- In Spring, the container is accessed through ApplicationContext.



TinyContainer reference: The container object is TinyContainer.
Access API: getBean(Class<T>)

What is ApplicationContext?

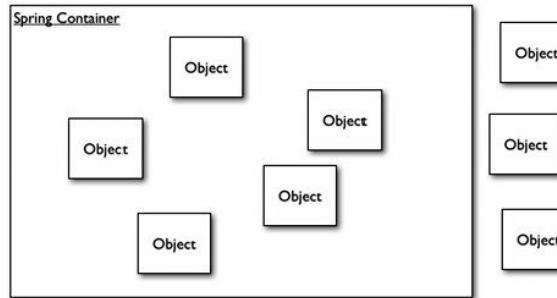
- ApplicationContext is the runtime container that holds what Spring knows how to build and what Spring has already created.
- It stores bean definitions, creates bean instances, and serves them when something needs a dependency.
- If a dependency is missing, startup fails because the container cannot produce that bean.

TinyContainer reference: ApplicationContext maps to TinyContainer.

Missing bean maps to NoBeanDefinitionException in getBean().

Beans Are Objects With a Contract

- A bean is an object created and the container decides when to create it, how to wire it, and when to run its lifecycle hooks.
- Domain objects (eg: new Student(...)) can still be created manually; component objects (eg: service, repository) are created by the container.
- Bean Definitions vs Bean Instances
 - A bean definition is the recipe: class, constructor, dependency requirements.
 - A bean instance is the real object created from that recipe and stored for reuse.



TinyContainer reference: Bean creation & storage in InstanceRegistry. See `InstanceRegistry.put()` and `InstanceRegistry.get()`.
DefinitionRegistry: BeanDefinition holds `implType` + constructor. | InstanceRegistry stores instances.

Where Do Beans Live?

- Beans live inside the container, not inside your code files.
- The container keeps references to created objects so it can reuse them and inject the same instance where required.
- When you ask for a dependency, you are asking the container for a reference it already owns.

TinyContainer reference: InstanceRegistry holds a map of Class -> Object.
getBean() returns after instanceRegistry.put(...).

How does the Container Chooses What to Create?

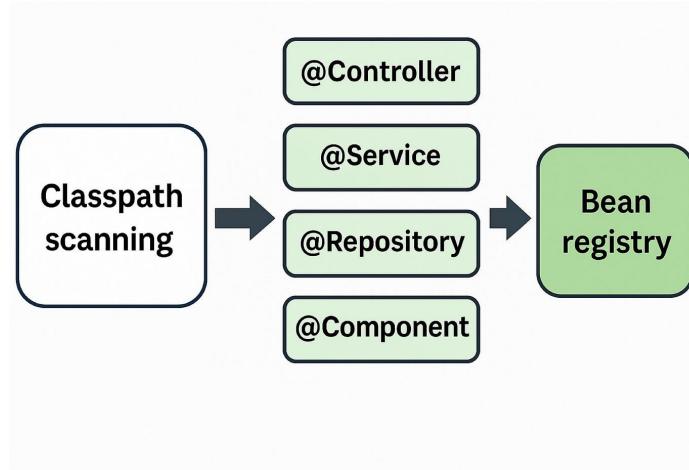
- Spring learns what to manage through configuration or by scanning packages for annotated components.
- Once it learns about a class, it records a bean definition for it.
- Missing annotations or missing scanning leads to missing bean errors during startup.

TinyContainer reference: container.register(X.class) in AppMain.java

Registration records a definition via definitionRegistry.put(...) in the register method inside TinyContainer

Component Scanning and Registration

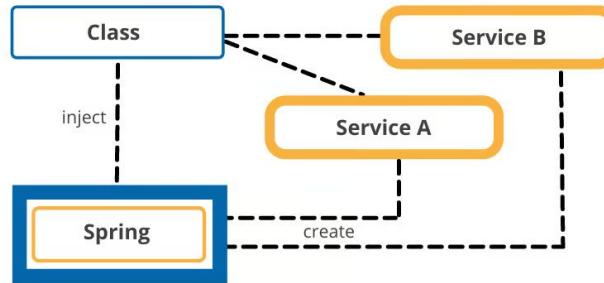
- Component scanning searches packages for classes marked as container-managed components.
- Spring registers bean definitions for those classes automatically.
- Practical model: scanning finds candidates, then registers them as definitions.



TinyContainer reference: We simulate post-scan by registering classes in AppMain.
DefinitionRegistry::keys shows what is registered.

Dependency Injection (DI)

- Dependency Injection means a class does not create its dependencies; it receives them.
- In Spring, constructor injection is standard because dependencies are explicit and required at creation time.
- DI follows from IoC because the framework can only wire objects if you stop wiring them yourself.
 - If you create new objects yourself, Spring does not inject their dependencies, so you must manually build and maintain the entire dependency chain (which brings back tight coupling)
 - You can also end up with duplicate instances (one Spring-managed, one manual), causing inconsistent behavior



TinyContainer reference: DI happens in `TinyContainer::createFromDefinition(...)`.

Dependencies resolved by `TinyContainer::getBean(depType)`. Injection happens at `ctor.newInstance(args)`.

How Dependencies Are Resolved

- When a constructor asks for an interface type, the container must pick a concrete implementation.
- If it finds none, you get a missing-bean error.
- If it finds more than one, you must disambiguate which one should be used.

TinyContainer reference: Implementation selection: `TinyContainer::resolveImplementation(requestedType)`.
None -> `NoBeanDefinitionException`. Many -> `IllegalStateException`.

Why Removing One Annotation Breaks Everything

- If a component is not registered, nothing can depend on it because the container cannot create it.
- The failure shows up where the container resolves a dependency, not necessarily where you forgot the annotation.
- Removing `@Service` in Spring produces a startup error for the first dependent bean.

TinyContainer reference: Equivalent: do not register a class in AppMain; startup fails inside `getBean()`.
Error when `definitionRegistry.get(implType)` returns null.

Lifecycle Means Creation Has Stages

- Lifecycle means a bean goes through stages, not just constructor runs.
- Typical lifecycle
 - **Instantiate:** Spring creates the object (calls the constructor).
 - **Dependencies injected:** Spring resolves required dependencies and supplies them (typically via constructor injection).
 - **Init callbacks:** Spring can run an “init step” after dependencies are injected, so the object can validate itself or prepare resources using those dependencies
 - Example: verify required settings are present, then open a connection to a database
 - **Ready for use:** the fully-initialized bean is stored in the container and can be injected into other beans and used by the application.
- Lifecycle exists because some setup must happen only after all dependencies are available.

TinyContainer reference: Instantiate: ctor.newInstance(args). Init stage: invokeInitMethods(instance). Ready after init and InstanceRegistry store.
@Init - com.guvi.module4.student.main.LifecycleProbe. Container: invokeInitMethods() calls @Init methods.

Circular Dependencies and Why Containers Detect Them

- A circular dependency occurs when A requires B and B requires A through constructors.
- Containers must detect this because there is no valid creation order.
- Spring fails fast during startup rather than producing partially wired objects.
 - **Fail Fast:** the app stops immediately at startup as soon as Spring detects a wiring/configuration problem, instead of continuing to run in a broken state and failing later at runtime (often with harder-to-debug errors)

TinyContainer reference: Cycle detection: creationStack inside getBean().

Failure type: CircularDependencyException.

Startup Hooks and Framework Executes Your Code

- Frameworks provide a way to run application code only after the container is fully ready.
- This exists because certain code only makes sense once dependencies exist.
- In Spring Boot, CommandLineRunner is one hook that runs after startup completes.
 - A hook is a specific spot in the framework's startup flow where it will run your code automatically (so you don't have to call it yourself).
 - CommandLineRunner is called right after startup finishes

TinyContainer reference: Warmup hook: StartupTask.

Runs in TinyContainer.start() by calling task.run(). | Demo: com.guvi.module4.student.main.DemoStartup.

What Happens During Startup

- Startup sequence
 - **Register definitions:** Record what components exist and how to build them (their “recipes”).
 - **Create objects:** Instantiate the components the container is responsible for.
 - **Resolve dependencies:** For each constructor parameter, fetch/create the required dependency and pass it in.
 - **Run init callbacks:** After dependencies are injected, run setup/validation methods that need those dependencies.
 - **Run startup hooks:** After the container is fully ready, run user code that should execute on startup (like a “run once after boot” step).
- The container performs the same work you used to do with *new keyword*, but centrally and consistently.
- When something fails in Spring, debug by checking: was the bean registered, created, wired, initialized.

TinyContainer reference: Register: AppMain calls container.register(...).

Create + wire: TinyContainer.start() calls getBean(...) per definition. | Init: invokeInitMethods(instance). | Hooks: StartupTask.run().

Debug: Use ContainerReport.print(container) to show definitions vs instances. Definitions vs instances printed using registry getters.

Spring Framework vs Spring Boot

- **Spring Framework** provides container fundamentals: ApplicationContext, bean definitions, DI, lifecycle.
- **Spring Boot** provides an opinionated startup layer: bootstrapping, conventions, sensible defaults, runtime wiring.
 - Spring Boot does not replace Spring fundamentals; it automates setup so you focus on application code.

TinyContainer reference: TinyContainer is a Spring-like core container.

A Boot-like layer would create the container, register components, then start it.

REST and APIs

REST and APIs

- Backend logic is only valuable when other software can call it reliably: web apps, mobile apps, internal tools, partner systems.
- A REST API exposes backend capabilities over the network using predictable URLs, HTTP methods, and consistent responses.
- Initial Goal: learn the shared vocabulary that lets any client talk to your Java backend.

Question: Name two different clients that could consume a Student backend API.

Answer: A browser-based admin dashboard and a mobile app can both call the same API.

What is an API?

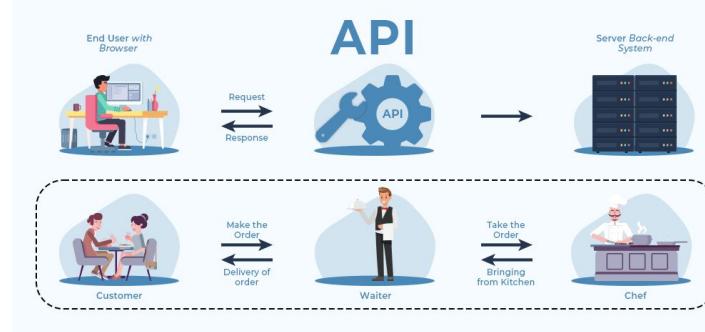
- **Application Programming Interface (API)** is a contract between a client and a server. Includes:
 - **Operations**: which actions are available on the server. Eg: create, read, update, delete
 - **Inputs**: what the client must send to the server
 - **Outputs**: what the client receives from the server
 - **Error meanings**: how failures are communicated from server to client
- A good API stays stable even when internal code changes, because clients depend on the contract.
- In this module, the contract is expressed using HTTP requests and HTTP responses.

Question: If we did not expose an API, how would a mobile app use the StudentService logic? Can we register a student from a mobile app?

Answer: It cannot call your Java methods over the network; the API is the network contract.

Why do APIs matter in real systems?

- APIs let independent teams and codebases integrate without sharing runtime or language.
- APIs keep your business logic stable for clients even when the backend implementation changes (refactors, new database, support new database).
- APIs force clarity: you define what data can be requested, what errors mean, and what the server guarantees.
 - In practice, these are often core interfaces or parent classes that are widely used in a project. So, changing a method definition can have cascading impact.

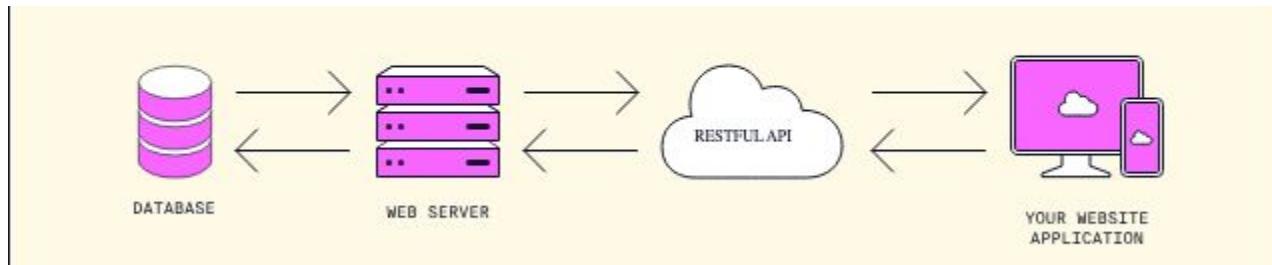


Question: Why is copying backend code into the frontend not a solution?

Answer: The frontend cannot safely own backend responsibilities like database access, authorization, and server-side validation.

What is REST?

- REST is an architectural style where the server exposes resources and clients manipulate them using standard HTTP methods.
- A resource is a domain concept like students or attendance records, identified by a URL path.
- REST is predictable because the same URL patterns and request types follow consistent rules across the API, instead of every endpoint behaving differently



Question: In a Student system, what is an example of a resource?

Answer: Students is a resource, and a specific student is a resource instance.

REST vocabulary: resource and representation

- A **resource** is the conceptual thing the server manages, such as a student record.
- A **representation** is the data format used to describe the resource (eg: JSON)
- Clients interact with representations, not Java classes
- On the flip side, the server maps internal models to external representations.
- Example: below is a presentation of a person

```
{  
  "id": "3f0c2a8c-7c55-4a3a-9d6a-4a1cf2c38f52",  
  "name": "Rahul",  
  "email": "rahul@gmail.com"  
}
```

Question: Why do we say representation instead of the Student object?

Answer: Clients receive JSON, not your Java object, and you can change internal classes without changing the representation.

REST URL: Design rules

- URLs should represent resources as nouns, not actions as verbs.
- Collections use plural nouns. Eg: retrieve ALL students from the server: /students.
- A single resource typically uses an ID. Eg: retrieve a single student: /students/{id}
 - {id} is called a *path parameter*
- Use the HTTP method to express the action
 - Eg: avoid URLs like /createStudent or /deleteStudent.
- Query parameters are for filtering and pagination, not for identity.
 - Eg: /students?sort=ascending&limit=10
- Good: GET /students/42. Poor: GET /getStudentById?id=42. Poor: POST /createStudent.

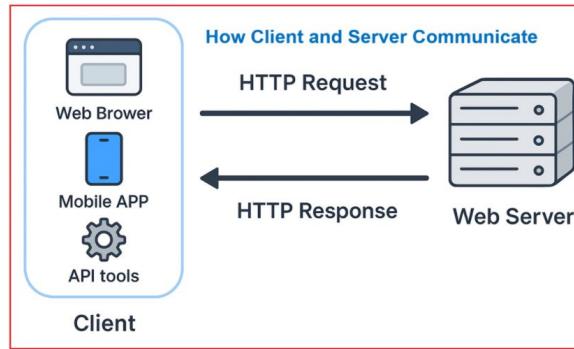
Question: Which is more REST-aligned and why: /createStudent or /students?

Answer: /students is better because the URL represents the resource; the action is expressed by the HTTP method.

HTTP Foundations

What's HTTP and why does REST use it?

- HTTP is the standard request-response protocol used by browsers and most software clients to talk to servers.
- HTTP defines a shared structure: request line, headers, optional body; response status, headers, optional body.
- REST uses HTTP because its method semantics and status codes give widely understood meaning without inventing a custom protocol.



Question: What is one advantage of HTTP being a standard protocol?

Answer: Universal tooling and debugging: all major browsers support it. Debugging? we can use Developer Tools in any browser

Why do HTTP methods exist?

- HTTP methods standardize what the client is trying to do without inventing custom rules.
- If every API used only one method for everything, clients could not reliably infer intent or expected behavior.
- Methods support correct system behavior
 - Read-only operations should be safe
 - Create/update/delete should signal state change

HTTP methods and what each one guarantees

- GET retrieves data and should not change server state.
 - GET /students
 - GET /students/{id}
- POST creates a new resource under a collection resource.
 - POST /students
- PUT updates the resource at a known identity.
 - PUT /students/{id}
- DELETE removes the resource at a known identity.
 - DELETE /students/{id}

Question: Which method should never be used to create new records and why?

Answer: GET, because it is expected to be read-only and safe to repeat without side effects.

HTTP request structure

- A request includes a method, a path, headers, and optionally a body.
- Headers carry metadata such as content type and authentication tokens.
- The body contains data the client sends to the server, typically JSON for create or update.
- How does a simple HTTP request look?

```
POST /students HTTP/1.1
Host: localhost:8080
Content-Type: application/json

{
  "name": "Sarandee",
  "email": "asha@example.com"
}
```

Question: Which part of the request carries the student data?

Answer: The request body carries the student data as JSON.

HTTP response structure

- A response includes a status code, headers, and optionally a body.
- The status code tells the client whether the request succeeded and how it failed if it did not.
- The response body returns data for success or an error payload for failure.
- How does a simple HTTP request look?

```
HTTP/1.1 201 Created
```

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

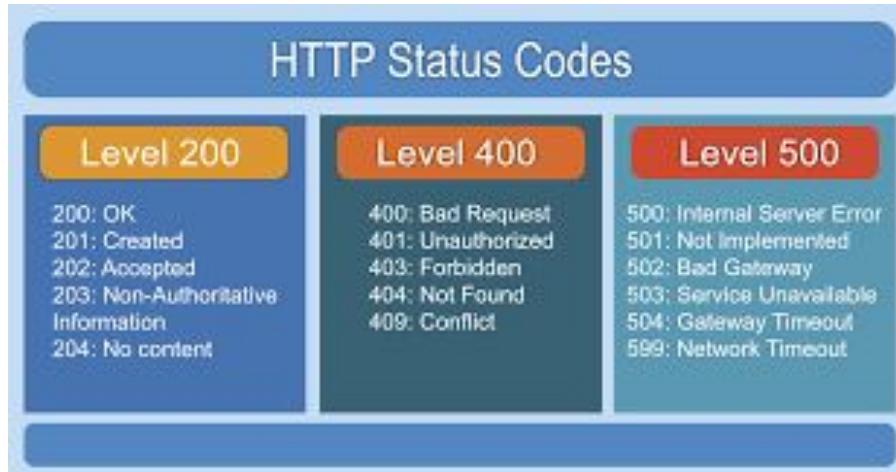
```
{  
  "id": "3f0c2a8c-7c55-4a3a-9d6a-4a1cf2c38f52",  
  "name": "Asha",  
  "email": "asha@example.com"  
}
```

Question: What is the difference between 201 and 200 in meaning?

Answer: 201 explicitly communicates that a new resource was created; 200 is a general success response.

Why do status codes exist?

- Status codes exist so clients can handle outcomes without guessing from text messages.
- They distinguish success from failure, and classify failures such as invalid input or missing resource.
- Consistent status codes reduce client-side bugs and make integrations easier to maintain.



Question: Why is it not enough to always return 200 with a message like failed?

Answer: Clients cannot reliably automate behavior; status codes are the standard machine-readable way to decide what to do next.

Status codes we will use in this module

- 200 OK indicates success for reads and updates.
 - GET /students/{id} succeeds
 - PUT /students/{id} succeeds
- 201 Created indicates a new resource was created.
 - POST /students succeeds
- 400 Bad Request indicates the request is invalid.
 - POST /students missing required field like email
- 404 Not Found indicates the resource does not exist.
 - GET /students/9999 when the id is not present

Question: If a student id does not exist, why is 404 better than 400?

Answer: The request format is valid, but the resource is missing; 404 communicates not found, not bad input.

Student API Design

Student API design

- Before writing code, define the contract:
 - what resources exist,
 - how clients identify them, and
 - what the server guarantees.
- A clean contract keeps the API stable even if implementation changes, such as moving from in-memory storage to a database.
- Today we design a Student API that is predictable, consistent, and easy for any client to use.

Question: If you change the URL or response format after clients start using your API, what breaks first?

Answer: Client code breaks first, because clients are coupled to the API contract.

Identify resources for the Student system

- Recall: a resource is a *domain concept the server manages*. For this API, the core resource is students.
- Students is a collection resource
- A single student is a resource instance identified by id.
- Related operations like attendance can be separate resources later; today we focus on students.
- Collection: /students. Single resource: /students/{id}.

Question: Why treat students as a resource instead of designing endpoints around methods like addStudent?

Answer: Because REST is resource-centered; the method is expressed by HTTP verbs, not baked into the URL.

Student endpoints final list

- List students
 - GET /students - retrieves the student list without changing server state.
- Get one student by id
 - GET /students/{id} - retrieves one student if the id exists.
- Create a student
 - POST /students - creates a new student under the students collection.
- Update a student
 - PUT /students/{id} - updates the student at a known id.
- Delete a student
 - DELETE /students/{id} - removes the student at a known id.

Question: Which endpoint should a mobile app call to show a Students list screen?

Answer: GET /students

Request body vs path variables vs query params

- Path variables identify a specific resource instance.
 - Example: GET /students/34 where 34 is the student id.
- Request bodies carry structured data for create or update.
 - Example: POST /students with JSON { "name": "...", "email": "..." }.
- Query parameters filter or shape a collection response without changing identity.
 - Example: GET /students?email=akash@example.com.

Question: Where should student id live: path variable, query param, or request body?

Answer: Path variable, because id is the identity of a single resource.

Response shape: success and error

- Responses should be consistent so clients can parse them reliably across endpoints.
- Success responses return a resource representation or a confirmation, depending on the operation.
- Error responses must be structured and actionable so clients can correct the request.

```
// Success example (GET /students/42)
{
  "id": "42",
  "name": "Geetha",
  "email": "geetha@example.com"
}

// Error example (invalid create)
{
  "error": "VALIDATION_ERROR",
  "message": "email is required"
}
```

Question: Why should error responses be structured JSON instead of plain text?

Answer: Clients need machine-readable errors so UI and integrations can handle failures consistently.

Spring Boot REST

Spring Boot REST building blocks

- Spring Boot exposes Java logic as HTTP endpoints without manual socket handling or manual HTTP parsing.
- Core building blocks:
 - controllers for HTTP entry points,
 - services for business rules,
 - repositories for data access.
- Spring container creates these objects, wires them, and calls controller methods when requests arrive.

Question: Which layer should contain business rules: controller or service?

Answer: Service, because controllers should stay focused on HTTP input/output, not business decisions.

Spring Framework vs Spring Boot for REST annotations

- Spring Framework provides the web framework and the annotation model for mapping HTTP requests to methods.
- Spring Boot configures Spring MVC by default when you add the web starter, so you do not manually assemble the web stack.
- In this session, the annotations you write are mostly Spring Framework; Boot handles startup, defaults, and auto-configuration.
- Spring Framework: `@RestController`, `@RequestMapping`, `@GetMapping`, JSON conversion integration via MVC infrastructure.
- Spring Boot: auto-configures Spring MVC and the embedded server so the API can run immediately.

Question: If you removed Spring Boot but kept Spring Framework, what would you lose first?

Answer: Boot auto-configuration and sensible-default startup; you would need more manual configuration to run the same web app.

What's RestController?

- `@RestController` marks a class as a web component that handles HTTP requests and returns data as the response body.
- It combines controller behavior with automatic response body handling, making JSON responses straightforward.
- Controllers translate HTTP requests into service calls and translate results back into HTTP responses.

```
@RestController  
@RequestMapping("/students")  
class StudentController {  
    // handler methods live here  
}
```

What's RequestMapping?

- `@RequestMapping` defines the base route for a controller or the route for a specific handler method.
- The final URL is built from the controller-level mapping plus the method-level mapping.
- This is how Spring selects the correct method for an incoming request path.

```
@RestController  
@RequestMapping("/students")  
class StudentController {  
  
    @RequestMapping("/{id}")  
    StudentDto getById(@PathVariable String id) { ... }  
}
```

Question: If the controller maps to /students and the method maps to /{id}, what is the full path?

Answer: /students/{id}

Request flow inside Spring Boot

- When an HTTP request arrives, the embedded server receives it and hands it to the Spring web layer.
- Spring matches the request path and method to a controller handler method using mappings.
- Spring constructs method arguments, calls the method, and converts the return value into an HTTP response.
- Flow: request arrives -> mapping chosen -> parameters extracted -> controller called -> return converted -> response sent.

Question: Where does business logic execute in this flow: server, controller, or service?

Answer: In the service; the controller should delegate business decisions to services.

How JSON becomes Java and Java becomes JSON

- Clients usually send JSON and receive JSON in REST APIs.
- Spring converts request JSON into Java objects and converts Java return values back into JSON.
- This avoids manual parsing and manual response construction, keeping controller code focused on intent.

Question: Why is automatic JSON conversion important for developer productivity?

Answer: It removes boilerplate parsing and response building, so developers focus on validation and business logic.

Activity

Activity: design the student endpoints

- Student fields: id, name, email.
- Design REST endpoints using correct HTTP methods and clean URLs.
- Specify method, path, request body if needed, and expected success status code.

Question: Write the five endpoints (method + path) for managing students.

Answer: GET /students, GET /students/{id}, POST /students, PUT /students/{id}, DELETE /students/{id}.

Activity answer key: endpoints and methods

- List students
 - GET /students
- Get student by id
 - GET /students/{id}
- Create student
 - POST /students with body { "name": "...", "email": "..." }
- Update student
 - PUT /students/{id} with body { "name": "...", "email": "..." }
- Delete student
 - DELETE /students/{id}

Question: Why is POST /students/{id} a bad idea for creating a student in this API?

Answer: The server should assign the id when creating; the client should not invent resource identities during creation.

Activity answer key: status codes and error cases

- Success cases
 - POST /students -> 201 Created
 - PUT /students/{id} -> 200 OK
 - DELETE /students/{id} -> 204 No Content
- Error cases
 - Missing required fields or invalid JSON -> 400 Bad Request
 - GET /students/{id} where id does not exist -> 404 Not Found
 - PUT /students/{id} where id does not exist -> 404 Not Found

Question: If a client sends invalid JSON, why is 400 better than 500?

Answer: 400 tells the client the request is wrong and must be fixed; 500 incorrectly suggests the server failed.

That's all for today! Questions?