Spring Boot

Topics

1) What is dependency injection? Why should I care?
2) How can Spring Boot give me a REST API?
3) Is handling errors hard?

Intro to Dependency Injection
&
Spring Boot

Dependency Injection (DI)

- Dependency Injection (DI)
  - ..
  - Separates..
    from..
- POJO
  ..
  - we'll differentiate this from using frameworks like Spring Boot

tightly coupled to a concrete class

loosely coupled, supporting polymorphism

DI Example

```
class AccountManager() {
    private Logger logger;
    private Database db;

    AccountManager() {
        logger = new Logger();
        db = new Database();
    }

    AccountManager(Logger logger, Database db) {
        this.logger = logger;
        this.db = db;
    }
}
```

Non-dependency injection:
class instantiates everything itself.

Dependency Injection:
Class is passed necessary objects.

- DI loosely couples classes:
  Client passes object in, so this class
  ..

What is Spring?

- Spring is..
  - To instantiate an AccountManager, we must have a reference to the Logger and Database to give it.
  - All parts of our code that instantiate an AccountManager need a logger and a database!
  - This can be burdensome!
- Instead, how about a "magic" way of saying: "Here's a Logger; please give it to every class wanting it"
  - That's what DI framework does.

What is DI Framework?

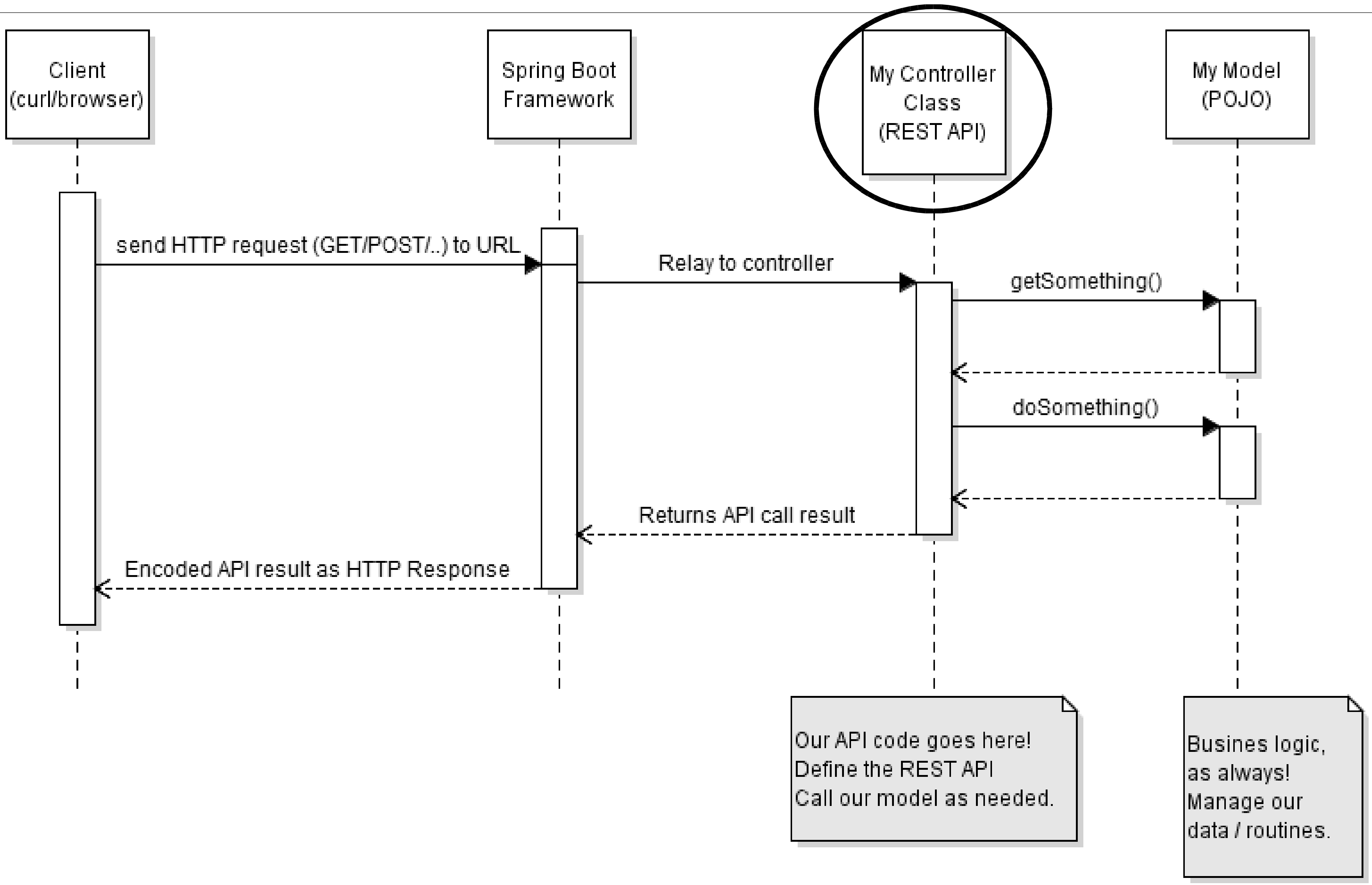
- DI Framework decouples our classes
  - the framework is told of objects to pass around (beans)
  - the framework instantiates our AccountManager class and passes in logger & DB (beans)
- Benefits of DI
  - ..
  - Easy to mock out objects for unit testing
- Benefits of DI Framework
  - creates the necessary object graph for us

What is Spring Boot?

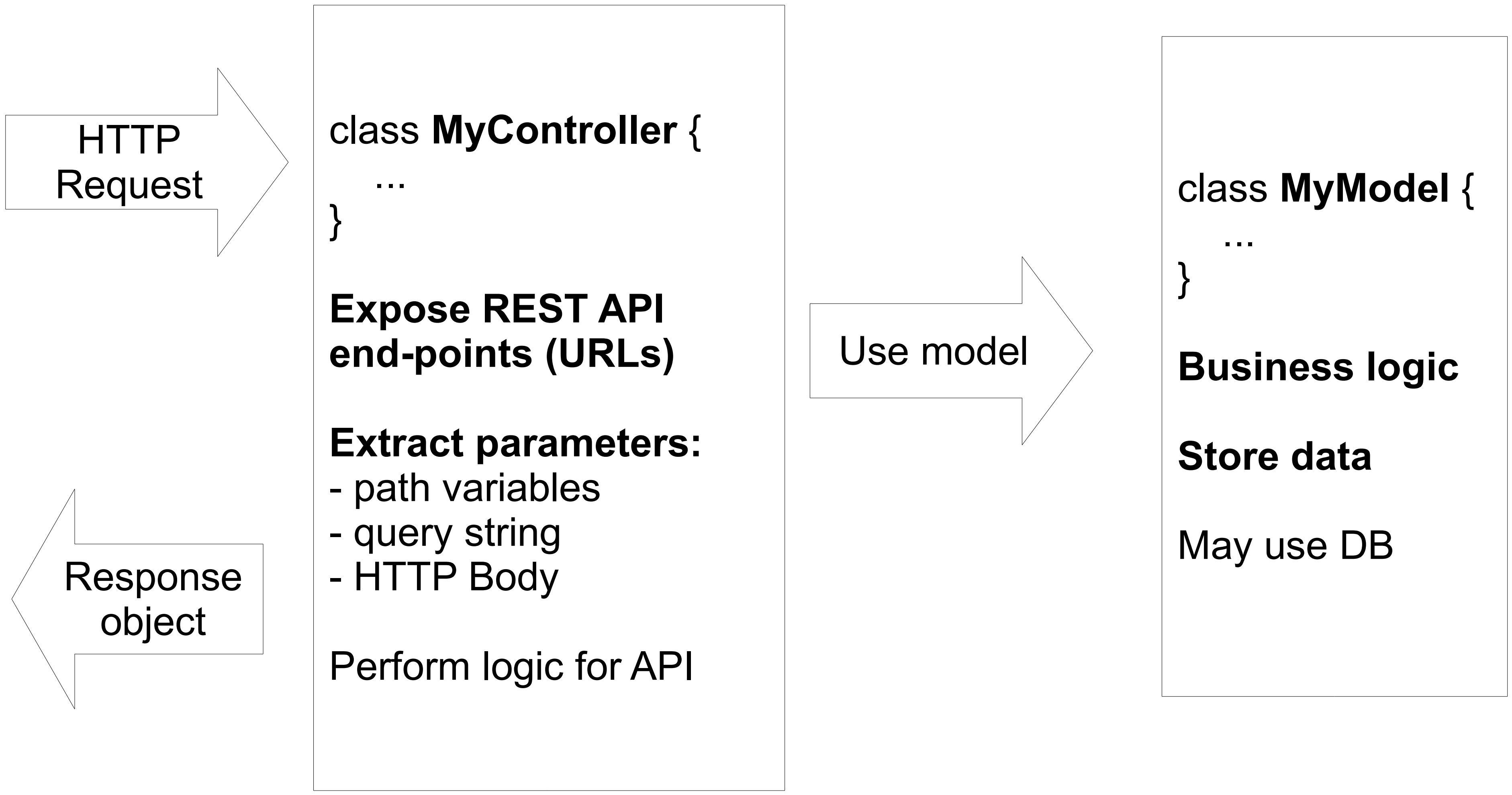
- What is Spring Boot?
  - It is a dependency injection framework with built in packages of functionality.
- Adds pre-configured packages to Spring
  - Easily add and configure DB, authentication, web, JSON, etc.
- Using Spring Boot feels a bit like magic:
  not just POJO!

REST APIs
with
Spring Boot

Back-end architecture



My Controller



class **MyController** {
    ...
}

Expose REST API end-points (URLs)

Extract parameters:
- path variables
- query string
- HTTP Body

Perform logic for API

class **MyModel** {
    ...
}

Business logic

Store data

May use DB

Spring Boot Hello World

- Demo: HelloWorld
  - No model; just a controller
  - GET / POST API via annotations
  - Parameter via body (POST)
- Usage
  - 1. View default message
    ```
    curl -s -i -X GET http://localhost:8080/greet
    ```
  - 2. Set 'name'
    ```
    curl -s -i -H "Content-Type: application/json" \
    -X POST -d 'Dr. Evil' http://localhost:8080/name
    ```
  - 3. See full Greeting
    ```
    curl -s -i -X GET http://localhost:8080/greet
    ```

Spring Boot Endpoint Annotations

- Creating an endpoint

```
@GetMapping("/minion")
public Minion getMinion() {
    return minion;                  // ‘minion’ just my field
}
```

  - Method name is irrelevant: think of it as a comment to the programmer
  - ..
    - all its public fields and public getters included.

Endpoint Arguments: Path

- Path variables to API specified in annotation

```
@GetMapping("/quotes/{id}")
public Quote getQuoteById(@PathVariable("id") long id) {

  for (Quote quote : quotes) {
      if (quote.getId() == id) {
          return quote;
      }
  }
  return null;

}
```

  - Can have multiple path variables in path (give each a unique name)

Endpoint Arguments: Body

- HTTP body comes to us as an object:

```
@PostMapping("/name")
public String getName(@RequestBody String name) {
    this.name = name;
    return name;
}
```

  - Commonly used for POST / PUT

Endpoint Argument: Query String

- For a GET you can support query strings:

```
@GetMapping("/quotes/")
Quote foo(
   @RequestParam(value="search", defaultValue="") String strSearch,
   @RequestParam(value="location", defaultValue="") String strLocation
) {
      System.out.println("Searching for " + strSearch
                        + " in location " + strLocation);
      ...
      return new Quote(....);
}
```

- Arguments in headers also possible, but not covered.

Demo

- Demo Quote Tracker
  - Show end points
  - Demo with curl

- Changes
  - Move Quote into a new model package
  - Add a QuoteManager class (POJO)
    - Move much of the logic from controller into QuoteManager class (in model)

MVC vs RESTful API

- MVC: Model View Controller
  - MVC in a web app: the server builds fully formed HTML web pages to transmit to the browser
- RESTful API
  - Client queries server endpoints for data
  - Client and server transmit JSON objects
  - With RESTful API server doesn’t generate HTML!
- Either way, dev team has to create the client
  - RESTful API is more flexible because it can be used by many clients (mobile, web, test scripts, ...)

HTTP Response Codes
&
Error handling

HTTP Response Codes

- API methods send HTTP 200 (OK) by default.
- Can change function to send specific code:

```
@PostMapping("/quotes")
@ResponseStatus(HttpStatus.CREATED)
public Quote newQuote(@RequestBody Quote quote) {
  // Set new quote's ID
  quote.setId(nextId);
  nextId++;

  // Store quote
  quotes.add(quote);

  // Return full quote so user gets ID
  return quote;
}
```

Error Handling

- Use exceptions to indicate errors
  - Uncaught exceptions generate
    ..
  - Use..
    to generate other HTTP responses such as
    400 (bad request) or 404 (not found)

Error Handling – Custom Exceptions

- Create custom exception with HTTP status code

```
// Support returning errors to client
@ResponseStatus(code = HttpStatus.BAD_REQUEST)
static class BadRequest extends RuntimeException {
}
```

- Throw the custom exception

```
@PostMapping("/quotes")
public Quote newQuote(@RequestBody Quote quote) {
    // validate data
    if (quote.getPerson().isEmpty()) {
        throw new BadRequest("Person must not be empty");
    }
      ... // do something useful!
}
```

Error Handling Demo

- Demo
  - Change Quote Tracker to handle errors:
    Return 404 (File Not Found) when requesting an invalid ID on GET.
- Hint: Have exception handle a message
  - Use an exception similar to this:

```
@ResponseStatus(code = HttpStatus.BAD_REQUEST)
static class BadRequest extends RuntimeException {
    public BadRequest() {}
    public BadRequest(String str) {
        super(str);
    }
}
```

FYI: Return ResponseEntity

- Endpoints can have full control of HTTP response

```
@PostMapping("/quotes")
public ResponseEntity<Quote> newQuote() {
     // ...
     return ResponseEntity
               .status(HttpStatus.CREATED)
               .body(myNewQuote);
}
```

FYI: Assign code to exception

- Can assign an HTTP response code to an existing exception (such as IllegalArgumentException)
  - Useful if code throws exceptions you don’t control but you want to set the response code.

```
@ResponseStatus(value=HttpStatus.BAD_REQUEST,
                    reason="Invalid parameter")
@ExceptionHandler(IllegalArgumentException.class)
public void errorHandleIllegalArg() {
   // Nothing to do
}
```

Summary

- Dependency Injection (DI)
  - Pass an object the references it needs; don’t let it instantiate the objects itself.
- Spring Boot
  - A DI framework which provides packages of functionality.
- Spring annotations to create API
  - @GetMethod(“/path”), ...
- HTTP response codes
  - @ResponseStatus(HttpStatus.CREATED)
  - Custom exceptions with status codes