Quarto Sprint

Durante questo Sprint approfondiremo il passaggio da un'architettura monolitica che utilizza Spring con JSP ad un'architettura più moderna e separata che utilizza Spring Rest (backend) con Angular (frontend).

Architettura Monolitica (Spring + JSP)

- **Tutto in uno**: In una tipica applicazione monolitica, il frontend (JSP) e il backend (Spring) sono strettamente accoppiati e distribuiti insieme. L'applicazione funziona come un'unità coesa.

- **Semplicità iniziale**: Una configurazione monolitica è spesso più semplice da sviluppare e distribuire inizialmente, poiché tutto ciò di cui hai bisogno è contenuto in una singola codebase.

Limitazioni:

- La **scalabilità** può diventare problematica, specialmente quando si desidera scalare solo alcune parti dell'applicazione.

- Le modifiche richiedono la **distribuzione** di tutto il monolite, il che può essere lento e rischioso.

- L'accoppiamento tra frontend e backend può rendere **difficile l'adozione** di nuove tecnologie o l'aggiornamento di quelle esistenti.

Architettura Decouplata (Spring REST + Angular)

- **Separazione delle Responsabilità**: Il backend (Spring REST) si occupa della **logica di business** e dell'accesso ai dati, mentre il frontend (Angular) si occupa della presentazione e dell'**interazione con l'utente**. Questi due componenti interagiscono tra loro attraverso **API RESTful**.

- **Scalabilità**: Poiché frontend e backend sono separati, possono essere **scalati indipendentemente**. Se c'è un carico pesante sul backend, per esempio, solo quello può essere scalato senza toccare il frontend.

- **Flessibilità e Mantenibilità**: Le modifiche al frontend o al backend possono essere effettuate indipendentemente l'una dall'altra, **facilitando gli aggiornamenti**. Si possono adottare o aggiornare nuove tecnologie sul frontend o sul backend senza disturbare l'altro lato.

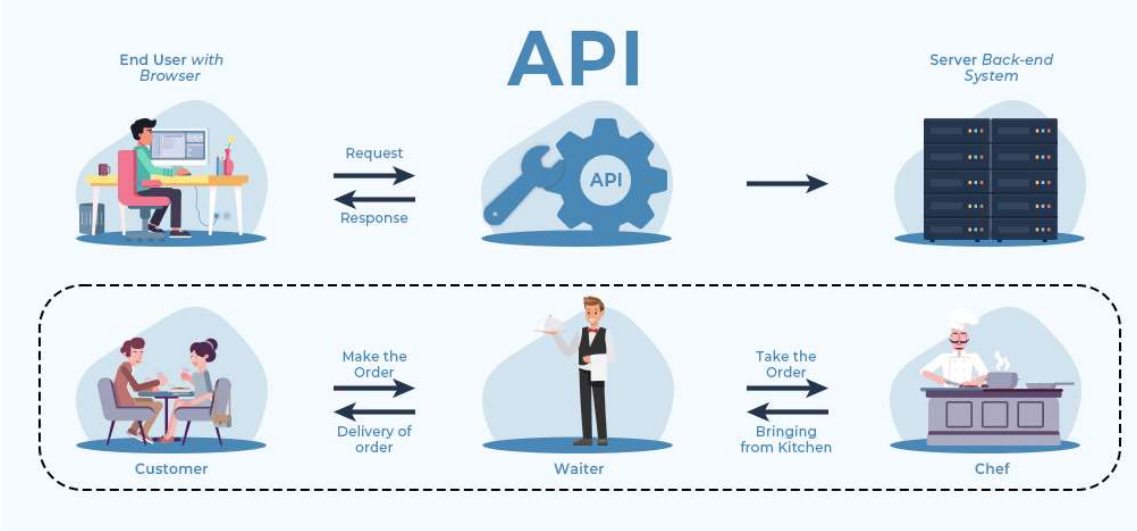
L’unico “svantaggio” se così lo possiamo chiamare riguarda la complessità iniziale maggiore dato che si ha a che fare con due progetti distinti. Sarà necessario gestire e mantenere la comunicazione tra frontend e backend.

Dunque durante questo Sprint svilupperemo 2 applicazioni, una per il backend e una per il frontend, che comunicheranno tra di loro grazie alle **API Restful**. Andiamo ora ad approfondire questo modello di comunicazione.

API (Application Programming Interface)

Un'API, o Application Programming Interface, è un insieme di regole e definizioni che permette a due applicazioni software di **comunicare** tra loro. È come un menu in un ristorante: il menu fornisce una lista di piatti che puoi ordinare, insieme a una descrizione di ciascun piatto. Quando specifichi quale piatto desideri, il ristorante (il sistema) prepara il piatto e te lo serve. In questo scenario, l'API è il menu.

Nel mondo del software, un'API fornisce una lista di comandi che possono essere utilizzati da un programma per **richiedere un servizio specifico da un altro sistema**, **applicazione** o **servizio**. Questi servizi possono variare dalla semplice richiesta di dati a operazioni complesse.



Tipologie di API:

- **API di Librerie o Framework**: Queste sono le funzioni e i metodi che un programmatore può utilizzare quando scrive codice per costruire applicazioni specifiche.
- **API Web**: Queste permettono la comunicazione tra diverse applicazioni su Internet.

REST (Representational State Transfer)

REST è un insieme di **principi architetturali,** basato sul protocollo http, utilizzati per la progettazione di networked applications. Fu introdotto e definito nel 2000 dalla tesi di dottorato di Roy Fielding, e da allora è diventato lo stile architetturale standard per la progettazione di API web.

Principi chiave di REST:

- **Stateless**: Ogni richiesta da un cliente a un server deve contenere tutte le informazioni necessarie per comprendere e elaborare la richiesta, senza conoscere lo stato delle chiamate precedenti. Dunque ogni chiamata è indipendente dalle altre.

- **Client-Server**: L'architettura client-server separa l'interfaccia utente dalle operazioni di storage dei dati, consentendo una maggiore scalabilità e flessibilità.

- **Cacheable**: Le risposte devono essere cacheable, ovvero i client devono poter immagazzinare risposte per migliorare le prestazioni.

- **Layered System**: Un sistema RESTful può avere più layer architetturali. L'uso di layer permette una maggiore modularità, e i client non necessitano di sapere oltre il loro layer di comunicazione.

- **Uniform Interface**: REST ha un set di vincoli che permettono una comunicazione standardizzata tra client e server, facilitando l'interazione tra componenti indipendenti.

Le API che aderiscono ai principi di REST sono chiamate **RESTful**.

Protocollo HTTP

Il protocollo HTTP (Hypertext Transfer Protocol) è un **protocollo di comunicazione** che funge da fondamento del World Wide Web. È utilizzato per trasferire dati tra il server web e il browser dell'utente. Concepito inizialmente per il trasferimento di documenti HTML, HTTP si è evoluto fino a diventare uno strumento adatto a trasferire vari tipi di dati, non solo documenti web, ma anche immagini, video, file JSON, XML e molto altro.

Vediamo le componenti fondamentali di una richiesta HTTP:

- **Metodo di Richiesta (HTTP Verb)**: Determina l'azione che il client vuole intraprendere sulla risorsa specificata. I metodi comuni includono:
    - **GET**: Recupera una risorsa.
    - **POST**: Invia dati al server per creare una nuova risorsa.
    - **PUT**: Aggiorna una risorsa esistente o ne crea una nuova.
    - **DELETE**: Rimuove una risorsa.
    - **HEAD**: Recupera solo gli headers della risorsa, senza il corpo.
    - **OPTIONS**: Restituisce i metodi HTTP supportati dalla risorsa specificata.
    - **PATCH**: Applica modifiche parziali a una risorsa.

- **URL** (Uniform Resource Locator): Identifica la risorsa sulla quale si desidera eseguire l'azione. Composto da:
    - **Schema** (ad es. http o https).
    - **Nome host** (ad es. www.example.com).
    - **Percorso** (ad es. /path/to/resource).

    - **Query string** (opzionale, ad es. ?key=value&key2=value2).

- **Versione HTTP**: Specifica la versione del protocollo HTTP utilizzato, come "HTTP/1.1" o "HTTP/2".

- **Headers di Richiesta**: Forniscono informazioni aggiuntive sulla risorsa da recuperare o su informazioni aggiuntive sulla richiesta stessa. Alcuni headers comuni sono:
    - **Host**: Specifica il dominio per la richiesta.
    - **User-Agent**: Identifica il software (ad esempio, il browser) che sta effettuando la richiesta.
    - **Accept**: Indica i tipi di media che il client può processare.
    - **Authorization**: Contiene le credenziali per l'autenticazione del client al server.
    - **Content-Type**: Indica il tipo di dati inviati nella richiesta (ad esempio, application/json).
    - **Content-Length**: Indica la lunghezza in byte del corpo della richiesta.

- **Corpo della Richiesta (Body)**: Contiene i dati che il client invia al server. Questo è tipico delle richieste POST e PUT (le chiamate GET e DELETE non possiedono un Body). Ad esempio, in una richiesta POST, il corpo potrebbe contenere dati di un modulo che un utente ha compilato su una pagina web.

Un esempio di richiesta HTTP potrebbe assomigliare a questo:

```
POST /path/to/resource HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: application/json
Content-Type: application/json
Content-Length: 57

{
  "key": "value",
  "anotherKey": "anotherValue"
}
```

Questo esempio rappresenta una richiesta POST al server www.example.com alla risorsa specificata nell'URL, inviando dati JSON come corpo della richiesta.

Spring Rest Controller

Dopo aver esplorato in profondità i principali elementi che costituiscono un'architettura basata su API RESTful, è ora di immergerci nel modo in cui questa architettura si manifesta all'interno del framework Spring. In particolare, ci focalizzeremo sul **confronto e le distinzioni** tra i tradizionali `**Controller**` che abbiamo esaminato nello sprint precedente e i `**RestController**` di Spring, pensati appositamente per la creazione e gestione delle API RESTful. Questa transizione ci fornirà una comprensione pratica e approfondita dell'evoluzione e dell'adattamento delle applicazioni Spring nell'era delle API.

In Spring, un REST Controller è una classe che assume il compito di gestire le richieste HTTP per fornire risposte in formato RESTful, solitamente in formato JSON o XML. Il framework Spring MVC, in combinazione con l'annotazione **@RestController**, rende piuttosto semplice creare API RESTful.

Ecco un'analisi dettagliata:

- **@RestController**: Questa è una combinazione di **@Controller** e **@ResponseBody**. L'annotazione indica a Spring che questa classe è un controller e che ogni metodo restituisce **direttamente la risposta al client** senza passare attraverso una vista (come ad esempio nello sprint precedente dove veniva restituita la JSP da visualizzare).

- **Definizione degli Endpoint**: Gli endpoint sono definiti all'interno di un RestController utilizzando le annotazioni specifiche per ogni verbo HTTP (@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, ecc.). Queste annotazioni aiutano a mappare specifiche azioni HTTP ai metodi del controller.

- **Path e Parametri**: Utilizzando @RequestMapping (o le annotazioni più specifiche come @GetMapping o @PostMapping), puoi specificare l'URL o il percorso dell'endpoint. Con **@RequestParam** puoi accedere ai valori dei parametri passati nell'URL della richiesta.

- **Corpo della richiesta**: Utilizzando **@RequestBody**, puoi legare il corpo della richiesta all'argomento del metodo del tuo controller. Questo è comunemente utilizzato con richieste POST e PUT.

- **Gestione delle risposte**: Il valore restituito dal metodo del controller (se non è void) viene **automaticamente** convertito nel formato appropriato (ad esempio, JSON) e inviato come **corpo della risposta HTTP**. E' possibile utilizzare l'oggetto **ResponseEntity** per avere un controllo più granulare sulla risposta, compresi header, codici di stato e corpo.

- **Gestione delle eccezioni**: Con **@ExceptionHandler**, puoi gestire le eccezioni a livello di controller, permettendo di restituire risposte personalizzate in caso di errori. @ControllerAdvice può estendere la gestione delle eccezioni su più controller, come abbiamo visto nello sprint precedente.

Il vantaggio dell'uso dei RestController in Spring è che il framework si occupa della maggior parte del lavoro pesante, come la serializzazione e la deserializzazione dei dati, consentendo agli sviluppatori di concentrarsi sulla logica dell'applicazione.

Vediamo un esempio:

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping("/getUser")
    public User getUser(@PathVariable Long id) {
        return userService.findById(id);
    }

    @PostMapping
    public User createUser(@RequestBody UserDTO user) {
        return userService.save(user);
    }

    @ExceptionHandler(UserNotFoundException.class)
    public ResponseEntity<String>
handleUserNotFound(UserNotFoundException ex) {
        return new ResponseEntity<>(ex.getMessage(),
HttpStatus.NOT_FOUND);
    }
}
```

Analizziamo le parti principali:

- **@RequestMapping("/api/users"):** Definisce il **percorso base** per tutte le API in questo controller. In altre parole, ogni endpoint in questa classe avrà un URL che inizia con /api/users.

- **@GetMapping("/getUser")**: Questo è un endpoint HTTP GET che viene utilizzato per ottenere le informazioni di un utente basate sull'ID fornito come parametro.
    - **@RequestParam("id") Long id**: Il parametro id viene passato attraverso la query URL. Ad esempio, una richiesta potrebbe avere un URL simile a /api/users/getUser?id=1.

- **@PostMapping("/insert")**: Questo è un endpoint HTTP POST utilizzato per creare un nuovo utente.
    - **@RequestBody UserDTO user**: La richiesta HTTP POST avrà un corpo contenente i dettagli dell'utente da creare. L'annotazione @RequestBody indica a Spring di legare il corpo della richiesta (tipicamente in formato JSON) all'oggetto UserDTO.

- **@ExceptionHandler(UserNotFoundException.class)**: Questa annotazione indica che il metodo sottostante dovrebbe essere invocato quando si verifica un'eccezione del tipo UserNotFoundException all'interno di questo controller.
    - **public ResponseEntity<String> handleUserNotFound(UserNotFoundException ex)**: Questo metodo restituisce una risposta personalizzata al client quando si verifica l'eccezione specificata. In questo caso, manda un messaggio d'errore e uno status HTTP di INTERNAL ERROR SERVER (500).

All'interno del nostro Controller, ciascun metodo rappresenta un endpoint di un API RESTful, a cui il frontend può inviare richieste per ottenere o manipolare informazioni. Ad esempio, se nel nostro frontend abbiamo la necessità di recuperare le informazioni dell'utente con id=1 dal database, effettueremo la seguente richiesta:

localhost:8080/api/users/getUser?id=1

Questa URL è strutturata in modo da puntare all'endpoint specifico /getUser all'interno del dominio localhost:8080, passando id=1 come parametro della richiesta.

Se tutto funziona come previsto e non si verificano errori, la risposta che otterremo sarà un oggetto JSON contenente le informazioni dell'utente con l'id specificato. Questo JSON rappresenta una forma strutturata e standardizzata per trasmettere dati tra il server e il client in modo efficiente e leggibile.

Postman

In molte fasi dello sviluppo, non è sempre fattibile o raccomandato testare le API direttamente attraverso l'interfaccia del frontend, soprattutto quando quest'ultimo non ha ancora integrato le nuove funzionalità previste. È qui che entra in gioco **Postman**.

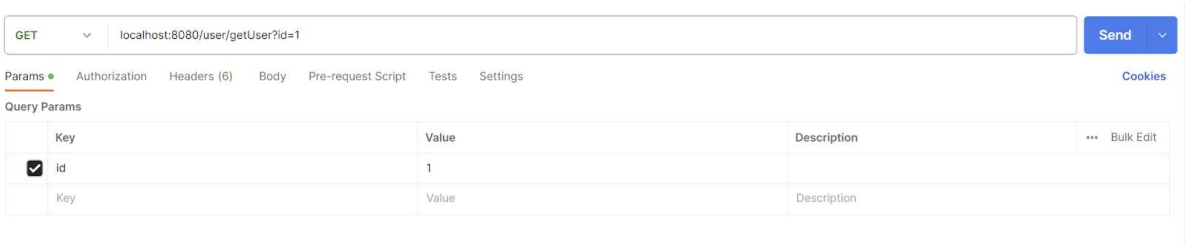
Postman è una rinomata piattaforma utilizzata dagli sviluppatori per progettare, testare e documentare API. Offre un'interfaccia intuitiva che facilita **la costruzione e l'esecuzione di richieste API**, permettendo di ricevere risposte in modo chiaro e organizzato. Questo strumento elimina la necessità di affidarsi a complesse righe di comando o alla creazione di codici ad hoc, semplificando e accelerando notevolmente l'interazione con le API.

Il setup di Postman è molto semplice, vi basterà scaricarlo al seguente link https://www.postman.com/downloads/ e una volta scaricato potrete già iniziare ad effettuare delle chiamate alle vostre API su Spring.

Vediamo degli esempi utilizzando UserController:

```
@RestController
@RequestMapping("/user")
@CrossOrigin(origins = "http://localhost:4200")
public class UserController extends AbstractController<UserDTO>{

	@Autowired
	private UserService userService;

	@PostMapping(value = "/login")
	public UserDTO login( @RequestBody LoginDTO loginDTO ) {
		return
userService.findByUsernameAndPassword(loginDTO.getUsername(),
loginDTO.getPassword());
	}

	@GetMapping(value = "/getUser")
	public UserDTO getUser( @RequestParam("id") Long id ) {
		return userService.read(id);
	}
}
```

Vediamo come effettuare la chiamata al metodo getUser passando un id come parametro tramite Postman:



Come prima cosa andiamo a selezionare il metodo http, in questo caso si tratta di una chiamata GET. Successivamente andiamo a inserire l'url della chiamata, in questo caso:

localhost:8080/user/getUser?id=1

Dove:

- **localhost**: L'hostname o il nome del **dominio**. In questo caso, "localhost" si riferisce al computer **locale** su cui viene eseguito il server. Di solito, si utilizza localhost durante lo sviluppo locale.
- **8080**: Questo è il numero della **porta** sulla quale il server sta ascoltando le richieste.
- **/user**: Questa parte rappresenta il **mapping** presente sopra UserController. Dunque serve a specificare che stiamo provando ad accedere ai metodi del controller User.
- **/getUser**: Questo è il nome del **metodo** che stiamo richiamando, in questo caso getUser.
- **?id=1**: Questa parte inizia con un punto interrogativo (?) e rappresenta i **parametri della query** che vengono inviati al server. In questo esempio, c'è un parametro di nome "id" con un valore di "1". I parametri della query sono utilizzati per fornire dati aggiuntivi nella richiesta. Se ci fossero più parametri, sarebbero separati da &. Ad esempio: ?id=1&name=Pippo.

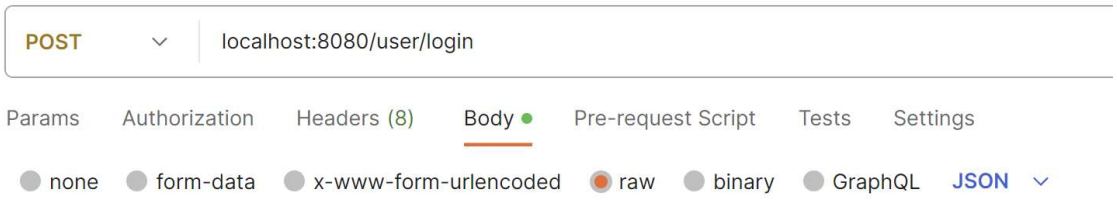
Una volta costruita la chiamata possiamo premere sul tasto "SEND" per inviare la richiesta.

In caso vada tutto bene otterremo una response con **Status Code 200** e un **JSON** contenente le informazioni dello **user** corrispondente all'id inserito nei parametri della richiesta.

Vediamo ora un esempio di chiamata POST provando a richiamare l'API /login.

```
@PostMapping(value = "/login")
      public UserDTO login( @RequestBody LoginDTO loginDTO ) {
            return
userService.findByUsernameAndPassword(loginDTO.getUsername(),
loginDTO.getPassword());

}
```

Nella definizione del metodo, troviamo l'annotazione @RequestBody. Questa specifica che il metodo si aspetta di ricevere dati all'interno del corpo della richiesta HTTP, in particolare, questi dati dovrebbero essere un JSON rappresentante un oggetto di tipo LoginDTO. Ecco come procedere per creare una tale chiamata:



Innanzitutto, impostiamo il metodo HTTP appropriato e forniamo l'URL corretto. Successivamente, è essenziale preparare il body della nostra richiesta. Per farlo:

1. Seleziona il tab "Body" nella tua interfaccia di creazione della richiesta.
2. Scegli l'opzione "raw" tra le varie scelte.
3. Dal menu a discesa sulla destra, seleziona "JSON". Questo stabilirà che il contenuto inserito sarà interpretato come JSON.

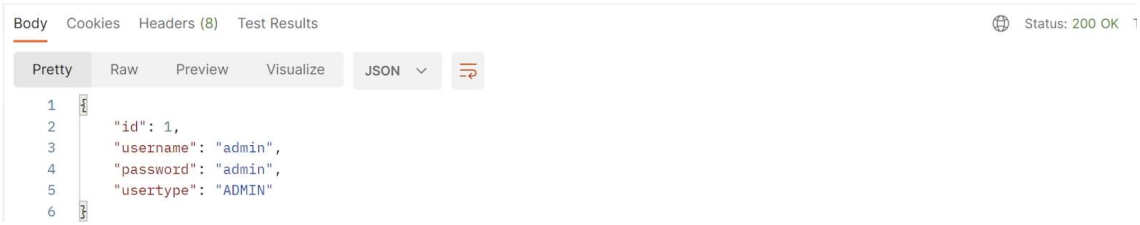
A questo punto, possiamo passare alla costruzione effettiva del nostro JSON. Prima di farlo, diamo un'occhiata alla struttura dell'oggetto LoginDTO per determinare quali attributi dovremmo includere:

```
public class LoginDTO {
   private String username;
   private String password;
}
```

Dalla classe sopra, vediamo che LoginDTO ha due attributi: username e password, entrambi di tipo String. Supponendo che si desideri eseguire il login con le credenziali username = admin e password = admin, il corpo JSON della nostra richiesta sarà costruito in questo modo:

```
{
   "username": "admin",
   "password": "admin"
}
```

A questo punto non ci resta che inviare la richiesta e verificare la risposta:



Come in precedenza abbiamo ottenuto una response con status code 200 e come corpo della response le informazioni dello user loggato.

Gestione degli errori

Nel costruire applicazioni robuste e resilienti, la gestione degli errori gioca un ruolo cruciale. In un'architettura basata su Spring Rest, gli errori possono sorgere per molteplici motivi: dati d'ingresso non validi, fallimenti nel backend, risorse non trovate, e così via. E' fondamentale dunque implementare una gestione degli errori adatta in modo tale da restituire risposte chiare e informative al client.
Per far ciò è importante seguire questi accorgimenti:

- **Custom Exception**: Creare eccezioni personalizzate specifiche per il dominio applicativo aiuta a identificare chiaramente il tipo di errore. Queste eccezioni possono estendere RuntimeException o Exception e fornire informazioni dettagliate sul motivo dell'errore.

- **Exception Handler**: Servono a centralizzare la gestione degli errori in modo tale da catturare eccezioni specifiche e di ritornare una risposta HTTP personalizzata al client.
- **Response Entity**: Utilizzando ResponseEntity, gli sviluppatori hanno un controllo completo sulla risposta HTTP, inclusi status, headers e body. È particolarmente utile per fornire messaggi di errore dettagliati e codici di stato appropriati.
- **Validazione dei Dati d'Ingresso**: Con l'annotazione @Valid e le annotazioni di validazione standard di Java (come @NotNull, @Size, ecc.), è possibile garantire che i dati forniti dal client rispettino certi criteri prima di essere processati.

Delineiamo in modo più chiaro il processo di gestione degli errori in un endpoint, utilizzando come riferimento l'endpoint /login presente nel UserController. Immaginiamo di voler fornire un feedback esplicito all'utente quando le credenziali fornite sono errate.

- Definizione dell'Eccezione Personalizzata: Innanzitutto, definiamo una nuova eccezione chiamata **InvalidCredentialsException** che eredita da RuntimeException. Questa eccezione servirà come segnale specifico per indicare credenziali non valide:

```
public class InvalidCredentialsException extends
RuntimeException{
    public InvalidCredentialsException(String message) {
        super(message);
    }
}
```

- Gestione Centralizzata delle Eccezioni: Implementiamo un gestore di eccezioni globale attraverso le annotazioni **@ControllerAdvice** e **@ExceptionHandler**. Questo approccio permette una gestione omogenea e centralizzata degli errori, facilitando la manutenzione e l'estensibilità:

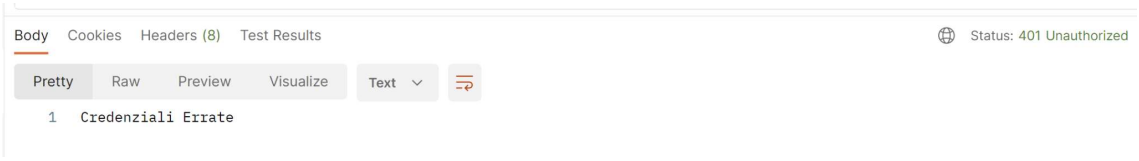
```
@ControllerAdvice
public class GeneralExceptionHandler {

    @ExceptionHandler(InvalidCredentialsException.class)
    public ResponseEntity<String>
handleInvalidCredentials(InvalidCredentialsException ex) {
        return new ResponseEntity<>(ex.getMessage(),
HttpStatus.UNAUTHORIZED);
    }

}
```

Con questo gestore, ogni volta che un'eccezione InvalidCredentialsException viene sollevata in qualsiasi controller, verrà catturata e gestita, restituendo un **messaggio di errore con uno status HTTP UNAUTHORIZED (401)**.

- Throw Exception: Non ci resta che **lanciare** la nostra eccezione nel caso in cui non venga trovato nessun utente corrispondente alle credenziali inserite:

```
    public UserDTO findByUsernameAndPassword(String username,
String password) {
          return
converter.toDTO(((UserRepository)repository).findByUsernameAndPasswo
rd(username, password)
                      .orElseThrow(() -> new
InvalidCredentialsException("Credenziali Errate")));
    }
```

A questo punto quando verrà effettuata una chiamata di login con credenziali errate, verrà sollevata un'eccezione InvalidCredentialsException da UserService, verrà "catchata" dal gestore di Exception che restituirà la seguente response:



Vediamo ora un esempio che riguarda la **validazione** dei parametri inseriti all'interno della request. Prendiamo come riferimento sempre l'endpoint di login. Mettiamo caso che lo username e la password presenti nella request debbano essere sempre presenti e non vuoti. Potremo implementare un controllo all'interno del metodo che verifichi che username e password siano diversi da null e che non siano stringhe vuote, lanciando un exception nel caso in cui la condizione non venga soddisfatta. Questa è una soluzione funzionante ma **SpringBoot offre delle annotation** che ci consentono di risolvere il problema in maniera più rapida ed elegante e risparmiando molto codice. Vediamo come:

1. Innanzitutto dovremo inserire all'interno del pom la seguente dipendenza:
   <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-validation</artifactId>
   </dependency>

   A questo punto potremo utilizzare le annotation di validazione di SpringBoot, in questo caso vedremo solamente **@NotBlank** e **@Valid**.

2. Successivamente andiamo nella classe LoginDTO e inseriamo l'annotation @NotBlank **sopra i campi** che dovranno essere notNull e notEmpty.

```
public class LoginDTO {

	@NotBlank(message = "Il campo username non può essere
vuoto")
	private String username;

	@NotBlank(message = "Il campo password non può essere
vuoto")
	private String password;
}
```

E' possibile specificare il messaggio da restituire in caso le stringhe non rispettino le condizioni precedenti.

3. A questo punto all'interno del metodo del Controller andiamo ad inserire l'annotation @Valid **affianco all'oggetto** che dovrà essere validato, in questo caso LoginDTO:

```
@PostMapping(value = "/login")
	public ResponseEntity<UserDTO> login(@RequestBody @Valid
LoginDTO loginDTO)
```

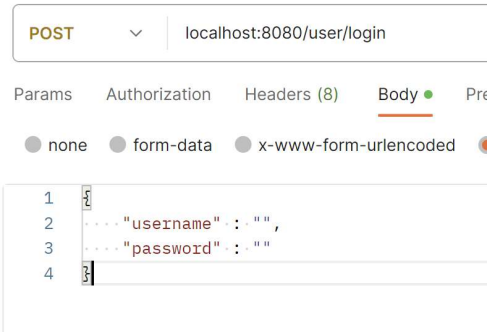
In questo modo diciamo a Spring che l'oggetto LoginDTO dovrà essere validato prima di essere passato al metodo, ovvero dovrà soddisfare le condizioni che abbiamo inserito all'interno della classe.

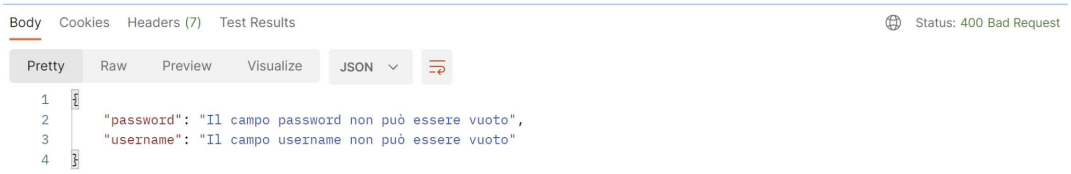
4. Nel caso in cui i campi della request non rispettino le condizioni verrà sollevata una **MethodArgumentNotValidException** che potremo andare a gestire con il nostro solito Exception Handler:

```
@ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String,String>>
handleMethodArgumentNotValid(MethodArgumentNotValidException
ex) {
        Map<String, String> errors = new HashMap<>();
        ex.getBindingResult().getAllErrors().forEach((error)
-> {
            String fieldName = ((FieldError)
error).getField();
            String errorMessage = error.getDefaultMessage();
            errors.put(fieldName, errorMessage);
        });
```

```
        return ResponseEntity.badRequest().body(errors);
    }
```

In questo caso ho realizzato un handler che si occupa di raccogliere gli errori dall'exception ed inserirli in una mappa in modo da raccogliere gli errori per ogni campo della classe che non ha rispettato le condizioni:





I messaggi di errore corrispondono a quelli inseriti nel parametro "message" dell'annotation @NotBlank

Esistono moltissime annotation di validazione dunque vi consiglio di andare ad approfondire.

Grazie a questi accorgimenti, quando l'utente tenta di accedere tramite l'endpoint di login con credenziali errate o non valide, riceverà una risposta chiara e concisa. Questa risposta facilita il lavoro degli sviluppatori frontend, che, oltre ad avere una chiara idea del tipo di errore che stanno ricevendo, possono ad esempio implementare un interceptor per visualizzare un avviso o una notifica all'utente basandosi sul codice di stato HTTP e sul messaggio di errore fornito.