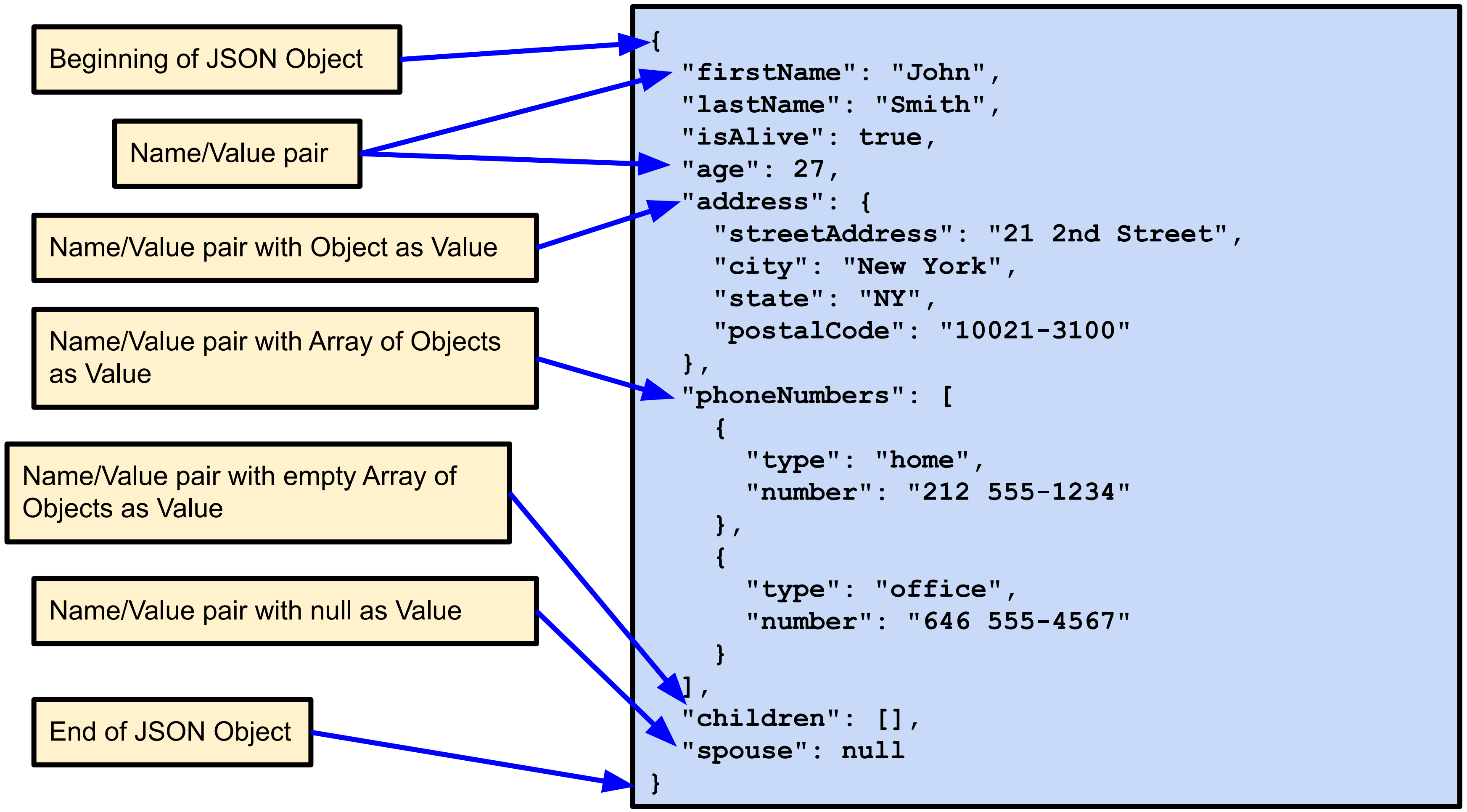
Server-Side APIs: Part 2

Let's Review JSON Syntax

JSON Data Types

- **Number**
  - Any number (integer or decimal)
- **String**
  - A sequence of zero or more Unicode characters. Strings are delimited with double-quotation marks and support a backslash escaping syntax.
- **Boolean**
  - Either of the values **`true`** or **`false`**
- **Array**
  - An ordered list of zero or more values, each of which may be of any type. Arrays use **`[]`** square bracket notation with comma-separated elements.
- **Object**
  - A collection of name–value pairs where the names (also called keys) are strings. Objects are delimited with **`{}`** curly braces and use commas to separate each pair, while within each pair the colon ':' character separates the key or name from its value.
- **`null`**


Beginning of JSON Object
Name/Value pair
Name/Value pair with Object as Value
Name/Value pair with Array of Objects as Value
Name/Value pair with empty Array of Objects as Value
Name/Value pair with null as Value
End of JSON Object

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

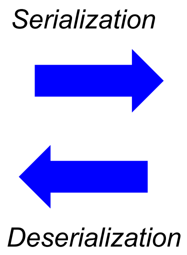
SERIALIZATION & DESERIALIZATION

Java Objects can easily be converted to JSON data and vice versa, Mapping from an Object to JSON is known as **serialization** of an Object. Mapping from JSON to an Object is known as **deserialization**.

Java

```
Review[];

public class Review {
    private int hotelID;
    private String title;
    private String review;
    private String author;
    private int stars;
}
```



JSON

```
[
    {
        "hotelID": 1,
        "title": "What a great hotel!",
        "review": "Great hotel,
        "author": "John Smith",
        "stars": 4
    },
    {
        "hotelID": 1,
        "title": "Peaceful night sleep",
        "review": "Would stay again",
        "author": "Kerry Gold",
        "stars": 3
    }
]
```

Introducing:
Inversion of Control & Dependency Injection

Inversion of Control

Inversion of Control refers to a principle that transfers the control of objects and their dependencies from the main program to a container or framework. In our case, we will be using ***Spring*** as the IoC container.

Benefits of Inversion of Control:

- Decouples the execution of a task from implementation.
- Allows a module to focus on the task it is designed for.
- Frees modules from assumptions about how other systems do what they do and instead relies on contracts.
- Prevents side effects when replacing a module.

Dependency Injection

Dependency Injection allows instances of classes that are depended on to be injected into a new object by Spring rather than being created by that object. This further decouples the classes from each other and allows the controller to be completely independent from any implementation of the DAO interface.

If you want to let Spring manage a class, you add an annotation to make Spring aware of the class. For the DAOs, you'd add an `@Component` annotation, which lets Spring be aware of and manage the class:

```
@Component
public class MemoryHotelDao implements HotelDao {
    ...
}
```

Object Creation vs. Dependency Injection

```java
// Using object creation

public class HotelController {

    private HotelDao hotelDao;
    private ReservationDao reservationDao;

    public HotelController() {
        this.hotelDao = new MemoryHotelDao();
        this.reservationDao =
                new MemoryReservationDao(hotelDao);
    }

    // other code


}
```

What we have been doing so far is creating instances of classes ourselves, either in the constructor using the **`new`** keyword or elsewhere in the code and providing them as params to the constructor when creating a class.

Object Creation vs. Dependency Injection

```
// Using object creation

public class HotelController {

    private HotelDao hotelDao;
    private ReservationDao reservationDao;

    public HotelController() {
        this.hotelDao = new MemoryHotelDao();
        this.reservationDao =
                new MemoryReservationDao(hotelDao);
    }

    // other code


}
```

```
@Component
public class MemoryHotelDao implements HotelDao {
    // other code
}
```

```
@Component
public class MemoryReservationDao implements
        ReservationDao {
    // other code
}
```

```
// Using dependency injection
@RestController
public class HotelController {

    private HotelDao hotelDao;
    private ReservationDao reservationDao;

    public HotelController(HotelDao hotelDao,
                           ReservationDao reservationDao) {
        this.hotelDao = hotelDao;
        this.reservationDao = reservationDao;
    }
    // other code
}
```

We annotate the implementation.classes of the DAOs with **`@Component`**, which allows Spring to manage them.

We include **`HotelDao`** and **`ReservationDao`** instances as parameters to the constructor.

Spring will search to see if it is managing **`HotelDao`** and **`ReservationDao`** instances and, if it finds them, it will inject them so that the params in the constructor refer to its managed instances.

We can assign those instances to our private variables so we can use them in our code.

Dependency Injection

This also makes the controller easier to unit test. When creating a new controller in a unit test, you can pass in a test version of the HotelDao:

```
HotelController testController = new HotelController(new TestHotelDao());
```

This allows the HotelController to depend on any HotelDao without being tied to a single implementation of it.

Validating Data

Server-Side Data Validation

It's important to handle data validation in back-end code and front-end code. Validation in Java is done using a Java standard called **Bean Validations**.

Bean Validations are annotations that are added to Java model classes and verify that the data in objects match a certain set of criteria. Here's an example:

```
public class Product {
    @NotBlank( message="Product name cannot be blank" )
    private String name;

    @Positive( message="Product price cannot be negative" )
    private BigDecimal price;

    @Size( min=20, message="Description cannot be less than twenty characters" )
    private String description;

    /*** Getters and Setters ***/
}
```

Server-Side Data Validation: @Valid

The validations that you add to your model objects aren't automatically checked for you. To check them in your Controller, you need to add the **`@Valid`** annotation to the model:

```
@RequestMapping(path = "", method = RequestMethod.POST)
@ResponseStatus(HttpStatus.CREATED)
public void create(@Valid @RequestBody Product product) {
    // data validation
    productsDao.add(product);
}
```

If the validation fails, meaning that the data supplied from the request doesn't pass the validation tests you defined in the model, then the response returns a status code of **`400 Bad Request`**, and the a JSON object containing info about the validation error is returned to the client.

Server-Side Data Validation: Bean Validations

Annotation	Applies To	Description
`@NotNull`	**Any**	The variable can't be `null`.
`@NotEmpty`	`array, List, String`	The variable can't be `null`. Also, if it's a `List` or `array`, it can't be empty. If it's a `String`, it can't be an empty `String`.
`@NotBlank`	`String`	The variable can't be empty or only contain white space characters.

Server-Side Data Validation: Bean Validations

Annotation	Applies To	Description
`@Min`	**`int, long,`**	The variable must have a value greater than the specified minimum. **`null`** values are skipped.
`@Max`	**`int, long`**	The variable must have a value less than the specified maximum. **`null`** values are skipped.
`@DecimalMin`	**`BigDecimal, double`**	The variable must have a value greater than the specified minimum decimal. **`null`** values are skipped.
`@Size`	**`array, List, String`**	The variable length, as an **`array`**, **`List`**, or **`String`**.

Server-Side Data Validation: Bean Validations

Annotation	Applies To	Description
`@Past`	`LocalDate`	The variable must represent a date in the past. `null` values are skipped.
`@Future`	`LocalDate`	The variable must represent a date in the future. `null` values are skipped.
`@Pattern`	`String`	The variable must hold a value that matches the specified regular expression. `null` values are skipped.

More About REST

REST stands for **Representational State Transfer** and is a series of guidelines for defining Web Services that are flexible and usable by a wide range of services.

- Uses all the standard web technologies you've already learned:
    - HTTP
    - URLs
    - JSON

- Building web services off of already widely adopted standards, REST makes it easy to tie APIs into existing applications.

More About REST

REST APIs are based on the concept of **resources** and building addresses (in the form of URLs) and actions (in the form of HTTP methods) on those resources. It also uses HTTP Status Codes to alert the API users about the results of the call

Resources are the objects defined in the application (i.e. Hotel, Reservation). You access resources through URLs, which specify the resource and, depending on which HTTP method you use, retrieve and modify them.

Addressing Resources

Goal	URL
Access a single resource	/products/342333
Access a resource that belongs to another resource	/products/342333/reviews
Access a particular review for a given resource	/products/342333/reviews/5674

CRUD Operations

***C*reate:**	`POST`
***R*etrieve:**	`GET`
***U*pdate:**	`PUT`
***D*elete:**	`DELETE`

Handling PUT Requests

```java
/**
 * Updates a product based on the ID and the request body
 *
 * @param product the updated product
 * @param id the id of the product that is getting updated
 */
@RequestMapping( path = "/products/{id}", method = RequestMethod.PUT )
public void update(@RequestBody Product product, @PathVariable int id) {
    product.setId(id);

    // Update product in underlying datastore
}
```

Handling DELETE Requests

```
/**
 * Removes a product based on the ID
 *
 * @param id the ID of the product to remove
 */
@RequestMapping( path = "/products/{id}", method = RequestMethod.DELETE )
public void delete(@PathVariable int id) {

    // Removes the product in underlying datastore

}
```

HTTP Status Codes: Success (2xx)

HTTP Status Code	Status Text	Usage
200	OK	Everything worked as expected—should have data returned. Common for **`GET`** requests.
201	Created	New resource was created. Common for **`POST`** requests.
204	No Content	Everything worked but no data is returned. Common for **`DELETE`** and **`PUT`** methods.

HTTP Status Codes: Client Error(4xx)

HTTP Status Code	Status Text	Usage
400	Bad Request	The request from the front-end had errors. Check the data passed back to see more specifics about the error. Often a data validation problem.
401	Unauthorized	The user isn't allowed to perform this action either because they aren't logged in or because their login information is wrong..
403	Forbidden	The logged in user isn't allowed to perform this action because they don't have permission.
404	Not Found	The given URL doesn't point to a valid resource.
405	Method Not Allowed	The HTTP Method given isn't valid for this URL. This could be because certain resources can't be updated or deleted and don't support the **`GET, PUT`**, **`DELETE`**, or **`POST`** methods.

HTTP Status Codes: Server Error(5xx)

HTTP Status Code	Status Text	Usage
500	Internal Server Error	The API itself has a problem and can't fulfill the request at this time. This could be due to a code issue or because services the API relies on, like databases or application servers, are down.

Returning Meaningful Status Codes

Normally, the correct status code is returned by default, but there may also be times when you want to return a different status code than the default. For instance, by default the status code **`200 OK`** is returned for a successful **`DELETE`**. However, REST suggests returning status code **`204 No Content`** on a successful **`DELETE`**. This is accomplished by adding the **`@ResponseStatus(value = HttpStatus.NO_CONTENT)`** annotation:

```
@RequestMapping( path = "/products/{id}", method = RequestMethod.DELETE )
@ResponseStatus(value = HttpStatus.NO_CONTENT)
public void delete(@PathVariable int id) {

    // Remove the product from underlying datastore

}
```

Returning Meaningful Status Codes

There may also be times when you want to return a different status code than the default due to an error. For example, if a user wanted to update the product with an ID of 13 and that ID wasn't in the database, you'd want to return a **`404 Not Found`** status code.

You can do that by setting up a special **`Exception`** that's linked to that status code with a **`@ResponseStatus`** annotation:

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class ProductNotFoundException extends Exception {}
```