

**CS 398: Application Development**

# Building Services

Web Services; REST; Spring Boot

# What is a service?

- A service is *software that provides capabilities to other software*.
- You have services on your computer that provide OS-level capabilities.
  - e.g. Unix daemon, Windows services for logging, messaging and so on.
- We're specifically going to talk about **shared services** running on a different computer. i.e. distributed systems.
  - Most applications work with online services in one form or another.
  - Cloud computing, SaaS.



# Why distribute services?

It can be useful to split processing across multiple systems:

- **Resource sharing.** We often need to share resources across users. For example, storing our customer data in shared databases that everyone in the company can access.
- **Reliability.** We might want to increase the reliability of our software by introducing redundant copies on different systems. This allows for fault tolerance i.e. failover.
- **Performance.** It can be more cost effective to have one machine running the bulk of our processing, while cheaper/smaller systems can be used to access that shared machine. It can also be cheaper to spread computation across multiple systems, running tasks in parallel. Distributed architectures provide flexibility to align the processing capabilities with the task to be performed.
- **Scalability.** If designed correctly, distributing our work across multiple systems can allow us to grow our system to meet high demand. Amazon for example, needs to ensure that their systems remain responsive, even in times of heavy load (e.g. holiday season).

# Distributed Architectures

# Client-Server

A **client-server architecture** is a model where a server provides capabilities to clients.

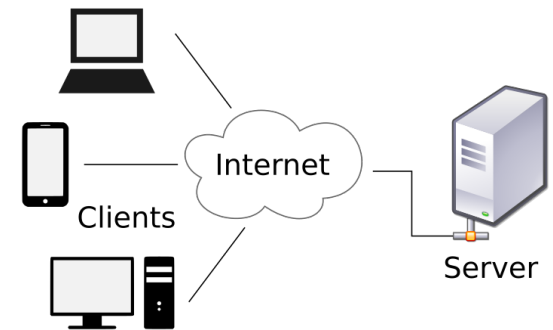
- **Client** – The process that issues a request to the second process i.e. the server. From the point of view of the user, this is the application that they interact with.
- **Server** – The process that receives and handles a request, and replies to the client.

## Advantages

- Separation of responsibilities i.e. presentation (client) from data (server).
- Reusability of server components and potential for concurrency (on a single server).
- Makes effective use of resources. i.e. promotes sharing.

## Disadvantages

- Limited server availability and reliability.
- Limited testability and scalability.
- Fat clients with presentation and business logic together.
- Limited ability to scale the server (need more processing? “buy a bigger server”).



# Multi-Tier

A **multi-tier architecture** (aka 2-tier, 3-tier or layered) separates an application into separate tiers or areas of concerns, often with each tier deployed on a separate computer.

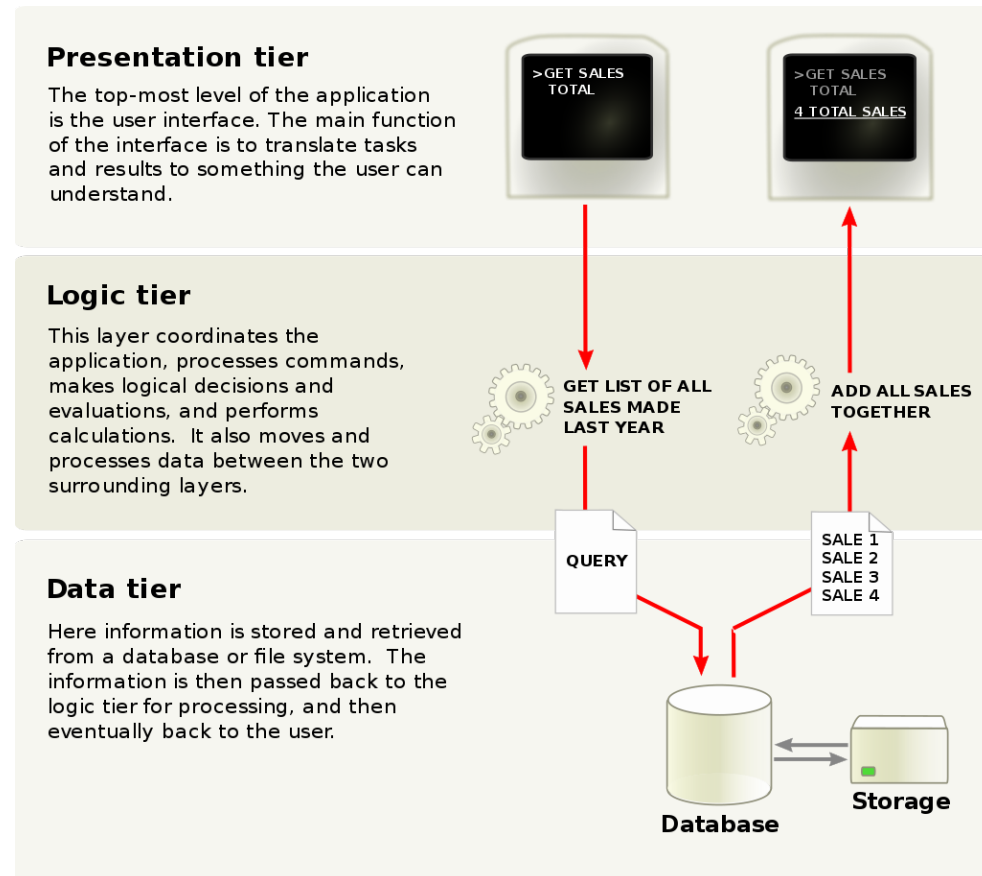
1. The top layer is the **Presentation tier**, which handles the UI logic. This is typically hosted on a client machine.
2. The middle layer is the **Application or Logic tier**, which handles “business logic”; the rules and state management of the application.
3. The bottom layer is the **Data tier**, which handles access, storage and management of the underlying data (i.e. database, or other).

## Advantages

- Enhances the reusability and scalability – as demands increase, extra servers can be added.
- Provides maintainability and flexibility.

## Disadvantages

- Greater complexity, more difficult to deploy and test across tiers.
- More emphasis on server reliability and availability.



Multi-tier. Often the user interface, business logic and data layers end up split apart.

# Service-Oriented Architecture

A [service-oriented architecture \(SOA\)](#) supports service orientation. A service is a discrete unit of functionality that can be accessed remotely and act independently to provide a service to a client. Services are loosely coupled but can work in coordination.

Principles of SOA, governing how services should be designed:

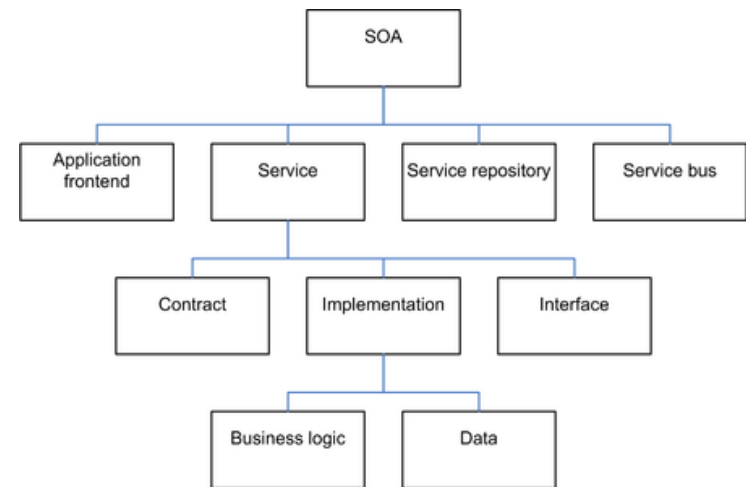
- **Service contract:** there should be an agreed upon interface for accessing a service.
- **Longevity:** services should be designed to be long-lived (long-running).
- **Autonomy:** services should work independently of one another.
- **Stateless:** services should not track state, but either return a resulting value or throw an exception.

Advantages

- A client can access services regardless of their platform, technology, language.
- Each service component is (mostly) independent from other services.
- The implementation will not affect the service as long as the interface is not changed.
- Support scalability by adding other services, but required manually redistribution.

Disadvantages

- Even more complexity in setting up a system. e.g. registry.
- Difficulty debugging, profiling and so on.



Service-Oriented-Architectures require a supporting infrastructure to run.

# Microservices

A [microservices architecture](#) arranges an application as a collection of loosely coupled services:

- Services are organized around business capabilities i.e. they provide specialized, domain-specific services to applications (or other services).
- Service are not tied to any one programming language, platform or set of technologies.
- Services are small, decentralized, and independently deployable.

A micro-service based architecture is really a subtype of SOA, which an emphasis on smaller, domain-specific components with very narrow functions.

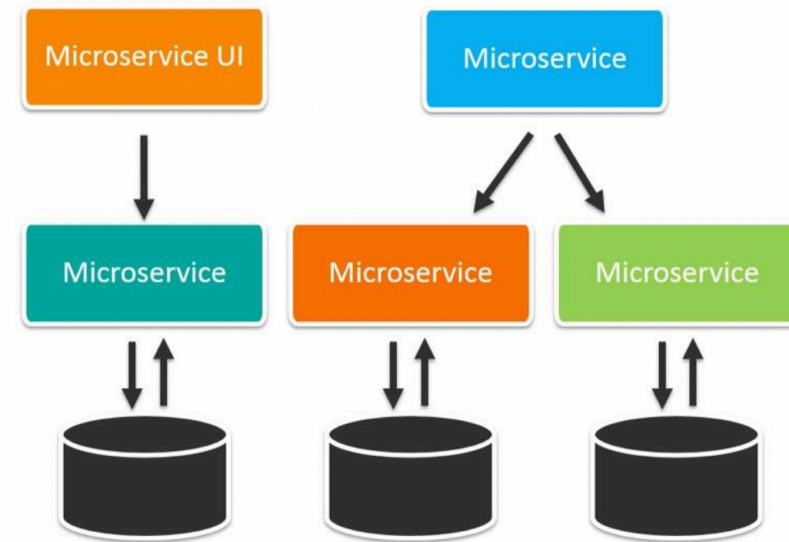
## Advantages

- Easier to design, build and deploy small targeted services.
- Redundancy - you can always “spin up” a replacement service.
- Performance - you can always “scale out” by firing up redundant services.

## Disadvantages

- Extremely difficult to test and debug.
- Practically requires supporting services, like a registry for processes to locate service endpoints.

## Microservices Architecture



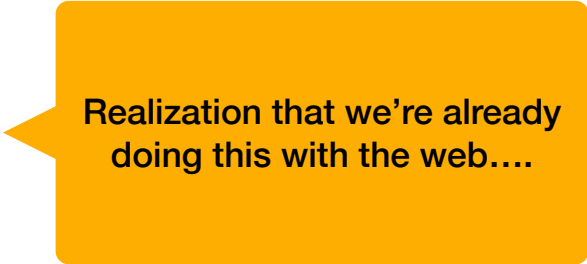
Microservices differ from regular services in that they are intentionally small, and easy to launch on-demand. They also rely less on brittle frameworks for coordination.



# Communication

# How do components communicate?

- Services of any type need to communicate with the client, and with one another.
- We need some protocols: how the data is structured, transmitted, validated.
- Many service architectures rely on heavyweight proprietary messaging protocols. e.g. RPC.
  - These protocols are often tightly coupled to the underlying implementation.
  - Early system focused on invoking remote functions - far too low level!
- The preferred approach, especially with microservices is to use a lightweight protocol.
  - Request-response model
  - Loose coupling between client and service
  - Well-formed, lightweight messages



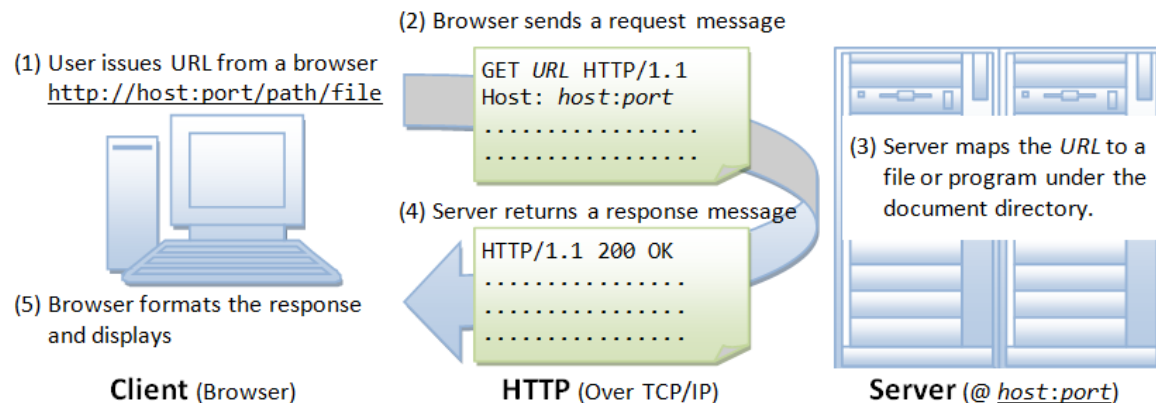
Realization that we're already doing this with the web....

# HTTP

The Hypertext Transfer Protocol (HTTP) is an **application layer** protocol that supports serving documents, and processing links to related documents, from a remote service.

HTTP functions as a **request-response** protocol:

- A **web browser** is a typical client, which the user is accessing. A **web server** would be a typical server.
- The user requests content through the browser, which results in an HTTP request being sent to the server.
- The server, which provides resources such as **HTML** files and other content or performs other functions on behalf of the client, returns a response message to the client that includes status, and possibly data.



# HTTP Request Methods

HTTP defines request methods to indicate the desired action, which will be taken on some resource (whatever the endpoint or URL represents). Often, the resource corresponds to a file or the output of an executable residing on the server.

- ★ **GET**  
The GET method requests that the target resource transfers a representation of its state. GET requests should only [retrieve data](#) and should have no other effect.
- **HEAD**  
The HEAD method requests that the target resource transfers a representation of its state, like for a GET request, but without the representation data enclosed in the response body. Uses include looking whether a page is available through the [status code](#), and quickly finding out the size of a [file](#).
- ★ **POST**  
The [POST method](#) requests that the target resource processes the representation enclosed in the request according to the semantics of the target resource. For example, it is used for posting a message to an [Internet forum](#), or completing an [online shopping](#) transaction.
- **PUT**  
The PUT method requests that the target resource creates or updates its state with the state defined by the representation enclosed in the request. A distinction to POST is that the client specifies the target location on the server.
- **DELETE**  
The DELETE method requests that the target resource deletes its state.

# Web services

A **web service** is a service that is built using web technologies, which serves up content using web protocols and data formats (e.g. JSON).

- Use HTTP as the basis for a more generalized service protocol that can serve up a broader range of data.
- Leverage the ability of web servers to handle HTTP requests in a very efficient manner, with the ability to scale to very large numbers of requests.

What we're missing are some guidelines that we can use on how to structure HTTP for generic requests, and make it useful for more than just a browser.

**REST**

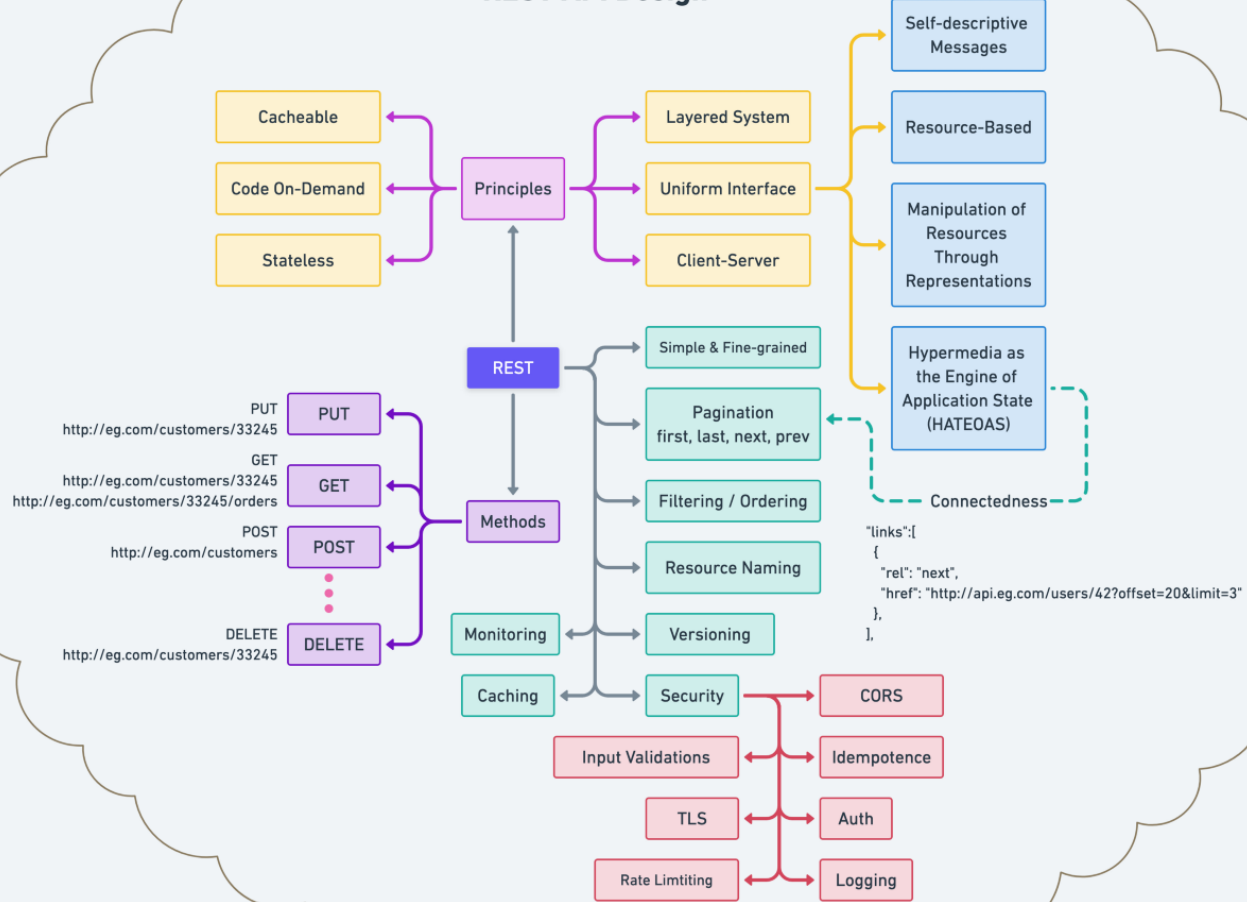
# REST

[REpresentational State Transfer \(REST\)](#), is a software architectural style that defines a set of constraints for how the architecture of an Internet-scale system, such as the Web, should behave.

- REST was created by [Roy Fielding](#) in his doctoral dissertation in 2000<sup>^</sup>.
- It has been widely adopted and is considered the standard for managing stateless interfaces for service-based systems.
- The term “RESTful Services” is commonly used to describe services built using standard web technologies that adheres to these design principles.

<sup>^</sup> Roy was also one of the principle authors of the HTTP protocol, and co-founded the Apache server project.

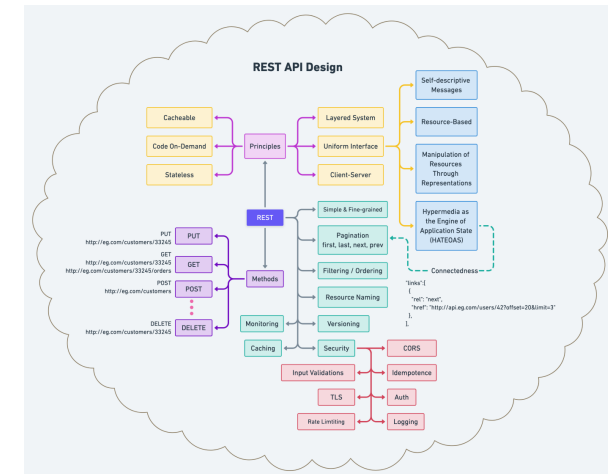
# REST API Design





# REST Principles

- 1. Client-Server.** By splitting responsibility into a client and service, we decouple our interface and allow for greater flexibility in how our service is deployed.
- 2. Layered System.** The client has no awareness of how the service is provided, and we may have multiple layers of responsibility on the server. i.e. we may have multiple servers behind the scenes.
- 3. Cacheable.** With stateless servers, the client has the ability to cache responses under certain circumstances which can improve performance.
- 4. Code On-Demand.** Clients can download code at runtime to extend their functionality (a property of the client architecture).
- 5. Stateless.** The service does not retain state i.e. it's idempotent. Every request that is sent is handled independently of previous requests. That does not mean that we cannot store data in a backing database, it just means that we have consistency in our processing.
- 6. Uniform Interface.** Our interface is consistent and well-documented. Using the guidelines below, we can be assured of consistent behaviour.



# Request Methods w. REST

For your service, you define one or more **endpoints** (URLs). Think of an endpoint as a function - you interact with it to make a request to the server. Examples:

- <https://localhost:8080/messages>
- <https://cs.uwaterloo.ca/asis>

To use a service, you format a request using one of these request types and send that request to an endpoint.

- **GET:** Use the GET method to read data. GET requests are safe and idempotent.
- **POST:** Use a POST request to STORE data i.e. create a new record in the database, or underlying data model. Use the request payload to include the information that you want to store. You have a choice of content types (e.g. multipart/form-data or x-www-form-urlencoded or raw application/json, text/plain...)
- **PUT:** A PUT request should be used to update existing data.
- **DELETE:** Use a DELETE request to delete existing data.

# API Suggestions

Here's some guidelines on using REST to create a web service [Cindrić 2021].

## 1. Use JSON for requests and responses

It's easier to use, read and write, and it's faster than XML. Every meaningful programming language and toolkit already supports it.

## 2. Use meaningful structures for your endpoint

Use nouns instead of verbs and use plural instead of singular form. e.g.

- GET /customers should return a list of customers
- GET /customers/1 should return data for customer ID=1.

# API Guidelines

## 3. Be Consistent

If you define a JSON structure for a record, you should always use that structure: avoid doing things like omitting empty fields (instead, return them as named empty arrays).

Good 👍

- GET /users
- POST /users
- GET /users/23
- PUT /users/23
- DELETE /users/23
- GET /users/23/comments

Bad 👎

- GET /user/23
- GET /listAllUsers
- POST /user/create
- PUT /updateUser/23
- GET /userComments/23

# Spring Boot

# What is Spring Boot?

**Spring** is a popular Java framework for building web services.

In the same way that JavaFX provides a framework for building GUIs on desktop, Spring provides a framework for building web services. It *greatly* reduces the effort required\*.

Spring is opinionated: it provides a strict framework and you are expected to add classes, and customize behaviour, to what is provided.

*Spring* is the framework, and *Spring Boot* is a set of tools for generating a project with all of the required dependencies. We'll walk through setting up a simple Spring Boot application. The steps are typically:

- Use Spring Boot to create a working project.
- Write controller, model, and other required classes.
- Write and run tests.

\* There are other frameworks, like [Ktor](#), that offer similar functionality. Spring is the most popular one.

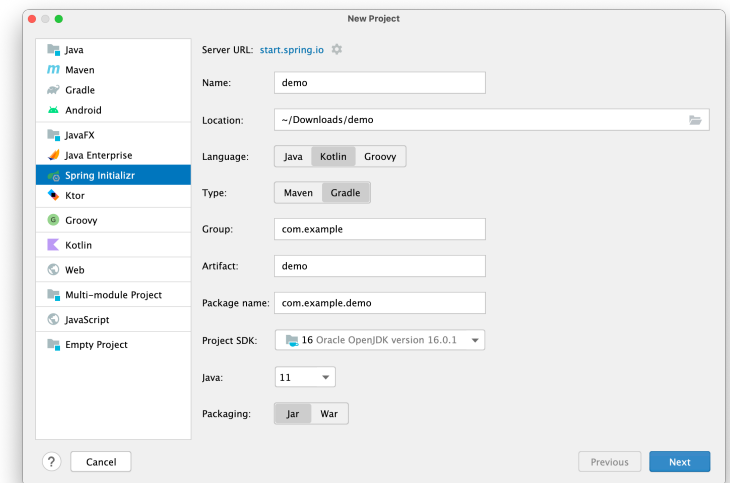
# Using Spring Boot

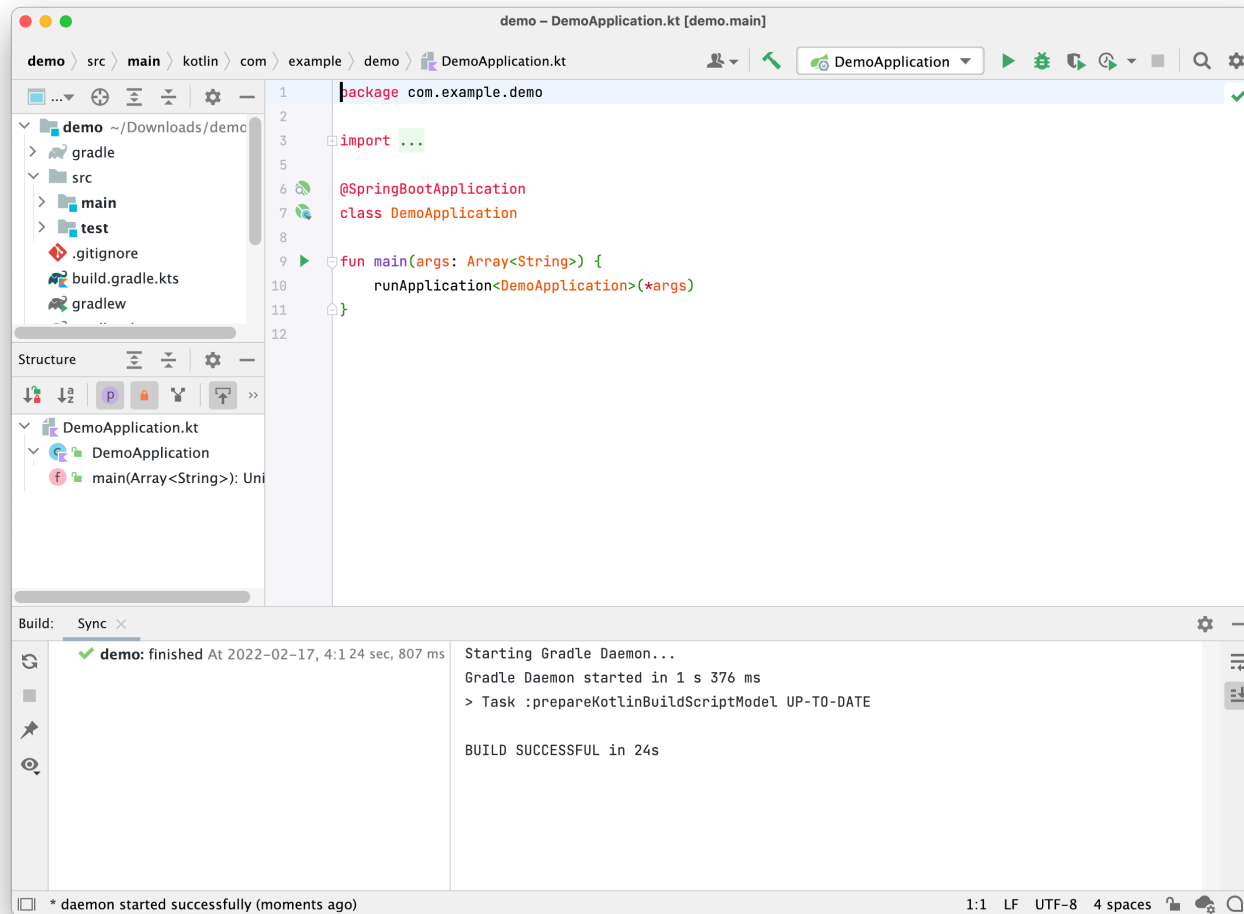
You can create a Spring Boot project in one of two ways:

- Visit [start.spring.io](https://start.spring.io) and set parameters for your project. Generate and download the project files.
- Alternatively, use the Spring project wizard in IntelliJ. Set the parameters for your project (Kotlin, Gradle) and follow instructions to generate an IntelliJ IDEA project.

Regardless of which you choose, you will be asked for dependencies. You will probably want to include these:

- **Spring Web:** this will embed a web server in your project so that you can easily test it.
- **Spring Data JPA:** this is an object-persistence layer that allows you to map classes to database tables and avoid writing SQL (JPA == Java Persistence API).
- **JDBC API:** this will allow you to use JDBC to access databases - helpful if your service needs to persist data.
- **H2 Database:** an embedded in-memory database for testing - you can also swap out for a different database if desired.





<https://localhost:8080>

A Spring Boot project looks like a normal project built on Gradle, but it's got a number of dependencies already imported and the framework is setup. It can be executed but won't do anything interesting yet.



# Adding a Controller

```
@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}

@RestController
@RequestMapping("/messages")
class MessageResource(val service: MessageService) {
    @GetMapping
    fun index(): List<Message> = service.findMessages()

    @PostMapping
    fun post(@RequestBody message: Message) {
        service.post(message)
    }
}

data class Message(val id: String, val text: String)

@Service
class MessageService {
    var messages: MutableList<Message> = mutableListOf()

    fun findMessages() = messages
    fun post(message: Message) {
        messages.add(message)
    }
}
```

```
/service/spring-server-basic
/service/spring-client
```

The controller manages the endpoints, and requests that are made to our service.

@annotations flag important classes

@RequestMapping sets the endpoint for this class

@GetMapping and @PostMapping are handlers for specific request types.

Our data class (we send and receive instances of this as JSON)

A placeholder service to hold data that we receive. POST will save a new message, GET will return the list of messages.

# Adding JPA Support

/service/spring-server-jpa  
/service/spring-client

```
@Service
class MessageService(val db: MessageRepository) {

    fun findMessages(): List<Message> = db.findMessages()

    fun post(message: Message){
        db.save(message)
    }
}

interface MessageRepository : CrudRepository<Message, String>{

    @Query("select * from messages")
    fun findMessages(): List<Message>
}

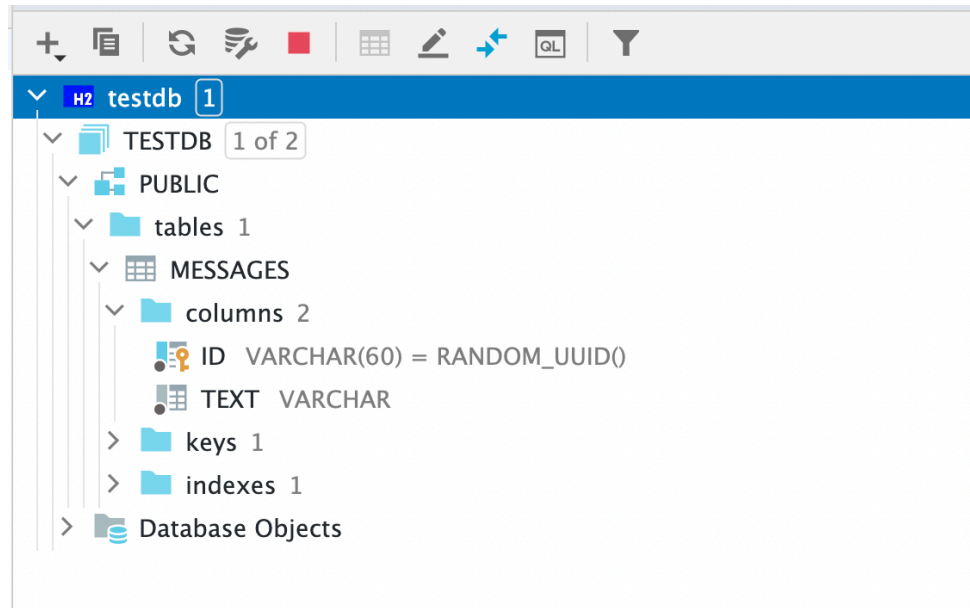
@Table("MESSAGES")
data class Message(@Id val id: String?, val text: String)
```

Dependency injection of a MessageRepository which the framework will manage.

A repository represents the data store. The framework will automatically use H2 and create methods for us to call. e.g. db.save

A table will also be created in the database, using the structure from our data class.

**JPA handles the object:database mapping**



JPA automatically created a table for our Repository, reflecting the structure of the Message data class.



# Testing your service

You can create and run HTTP requests from within IntelliJ IDEA.

1. Click on the drop-down menu beside your `@GetMapping` or `@PostMapping`.
2. Select “Generate request in HTTP Client”.
3. You should see a screen where you can format and run your requests.

```
@RestController
class MessageResource(val service: MessageService) {
    @GetMapping
    fun index(): I

    @PostMapping
    fun post(@Req
        service.post(message)
    }
}

### Post "Privet!"
POST http://localhost:8080
Content-Type: application/json

{
    "text": "Privet!"
}

### get all the messages
GET http://localhost:8080/
```

# Client Requests

# Making an HTTP Request

How do we actually use our service? Kotlin (and Java) includes libraries that allow you to structure and execute requests from within your application.

This example fetches the results of a simple GET request:

```
val response = URL("https://google.com").readText()
```

The `HttpRequest` class uses a builder to let us supply as many optional parameters as we need:

```
fun get(): String {  
    val client = HttpClient.newBuilder().build()  
    val request = HttpRequest.newBuilder()  
        .uri(URI.create("http://127.0.0.1:8080"))  
        .GET()  
        .build()  
  
    val response = client.send(request, HttpResponse.BodyHandlers.ofString())  
    return response.body()  
}
```

# Sending Data

Here's a POST method that sends an instance of our Message class to the service that we've defined, and returns the response. We use serialization to encode it as JSON.

```
fun post(message: Message): String {
    val string = Json.encodeToString(message)

    val client = HttpClient.newBuilder().build();
    val request = HttpRequest.newBuilder()
        .uri(URI.create("http://127.0.0.1:8080"))
        .header("Content-Type", "application/json")
        .POST(HttpRequest.BodyPublishers.ofString(string))
        .build()

    val response = client.send(request, HttpResponse.BodyHandlers.ofString());
    return response.body()
}
```

See the public repo for these samples, under `/service/sprint-client` and `/service/spring-server`.



# Resources

# Resources

- Jan Bodnar. 2022. Kotlin HTTP GET/POST request. <https://zetcode.com/kotlin/getpostrequest/>
- Vedran Cindrić. 2021. The 10 REST Commandments. <https://trebble.com/blog/the-ten-rest-commandments>
- Matt Greencroft. 2018. Building Spring Boot Applications with the Kotlin Programming Language. Manning Publishing. Video.
- JetBrains. 2021. Create a RESTful web service with a database using Spring Boot – tutorial. <https://kotlinlang.org/docs/jvm-spring-boot-restful.html>
- JetBrains. Kotlin Server Side. <https://kotlinlang.org/lp/server-side>
- Love Sharma. 2021. Principles & Best Practices of REST API Design. <https://blog.devgenius.io/best-practice-and-cheat-sheet-for-rest-api-design-6a6e12dfa89f>
- arjuna sky kok. 2021. Kotlin and Spring Boot: Getting Started. <https://www.raywenderlich.com/28749494-kotlin-and-spring-boot-getting-started>
- VMware. 2022. (Spring) Accessing Data with JPA. <https://spring.io/guides/gs/accessing-data-jpa/>
- VMware. 2022. Building web applications with Spring Boot and Kotlin. <https://spring.io/guides/tutorials/spring-boot-kotlin/>
- VMware. 2022. Spring Boot Reference Documentation. <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#legal>