framework training
We love technology

Introduction to Systems Integration and Continuous Integration

Informed Academy

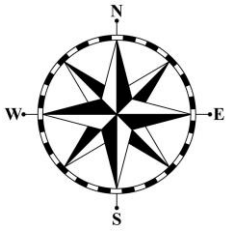
020 3137 3920 @FrameworkTrain
frameworktraining.co.uk

Welcome

Toby Dussek

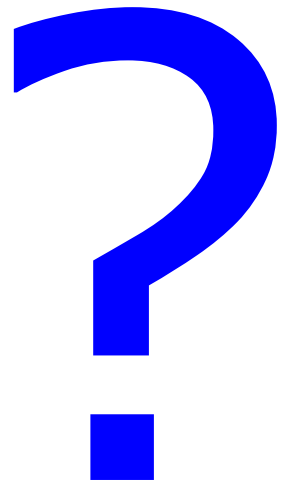
Informed Academy

Introduction



- Introduction to Systems Integration
  - RESTful services
  - Docker
  - GraphQL services
  - Security in S/W systems
  - Databases
  - Object Persistence / Object Relational Mappers
  - Hibernate and JPA
- Introduction to Continuous Integration tools
  - Jenkins

Questions?



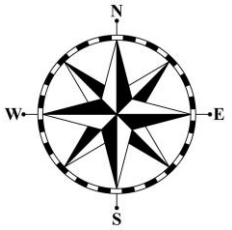
RESTful Services

Toby Dussek

Informed Academy

{ REST }

Plan for Session



- Web Services
- What is REST?
  - The REST Design pattern
  - Rest – Not a Standard
- Spring REST
- Building a Spring REST application
  - RESTFul Controller
  - Spring Application & Dependencies
  - More complex controller
- Introduce Spring REST Clients

Web Services

- Numerous ways of creating web services
  - JAX-WS (SOAP) based POJOs
  - Spring WS (SOAP-based Web Services)
  - JAX-RS based POJOs (RESTFul Web Services)
  - Spring RS POJOs (RESTFul Web Services)
  - AXIS 2/3 and CXF from Apache
  - Build it yourself …
- But which to use?

What is REST?

- **Re**presentational **S**tate **T**ransfer
  - Lightweight, resource-oriented, architectural style
- "REST" was coined by Roy Fielding in his Ph.D.
  - describes design pattern for implementing networked systems
  - Fielding – one of the principle authors of HTTP …
  - .. used REST to describe the architectural pattern of the web
  - See http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
- Interactions done using HTTP
  - *retrieve* information (HTTP GET)
  - *create* information (HTTP POST)
  - *update* information (HTTP PUT)
  - *delete* information (HTTP DELETE)

The Web is a RESTFul system

- HTTP provides a simple set of operations
- All Web exchanges done using simple HTTP API:
  - GET = "give me some info" (Retrieve)
  - POST = "here's some new info" (Create/Update)
  - PUT = "here's some update info" (Create/Update)
  - DELETE = "delete some info" (Delete)
- The HTTP API is CRUD
  - (Create, Retrieve, Update, and Delete)

REST - Not a Standard

- REST is not a standard
  - you will not see the W3C putting out a REST specification
- REST is just a design pattern
  - you can't bottle up a pattern
  - you can only understand it and design your Web services to it
- REST does prescribe the use of standards:
  - HTTP, URLs, JSON / XML / Text etc
- But may use tools to simplify processing of content,
  - e.g. libraries in different languages etc.

REST Example Resource URLs

- GET http://bookshop.com/books
  - retrieve information about all books
- GET http://bookshop.com/books/1234
  - get information for book with ISBN 1234
- POST http://bookshop.com/books
  - add new book; details in body of request
- PUT http://bookshop.com/books
  - update an existing book; details in body of request
- DELETE http://bookshop.com/books/1234
  - delete book with ISBN 1234

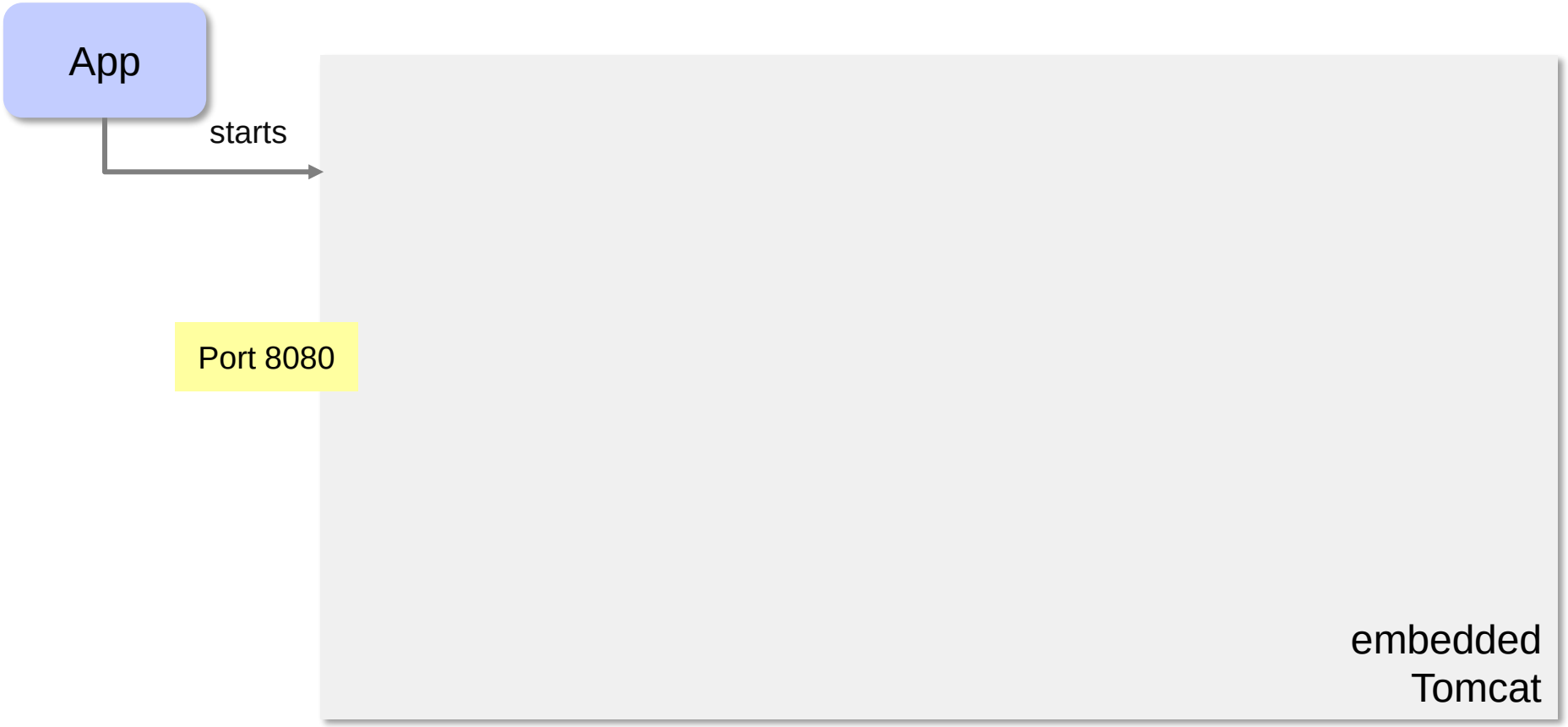
Spring REST

- Spring-REST
  - introduced back in Spring 3.0
- Provides first-class support for REST-style mappings
  - supports different HTTP request methods
  - extraction of URI template parameters
  - content negotiation in view resolver
  - in natural MVC style

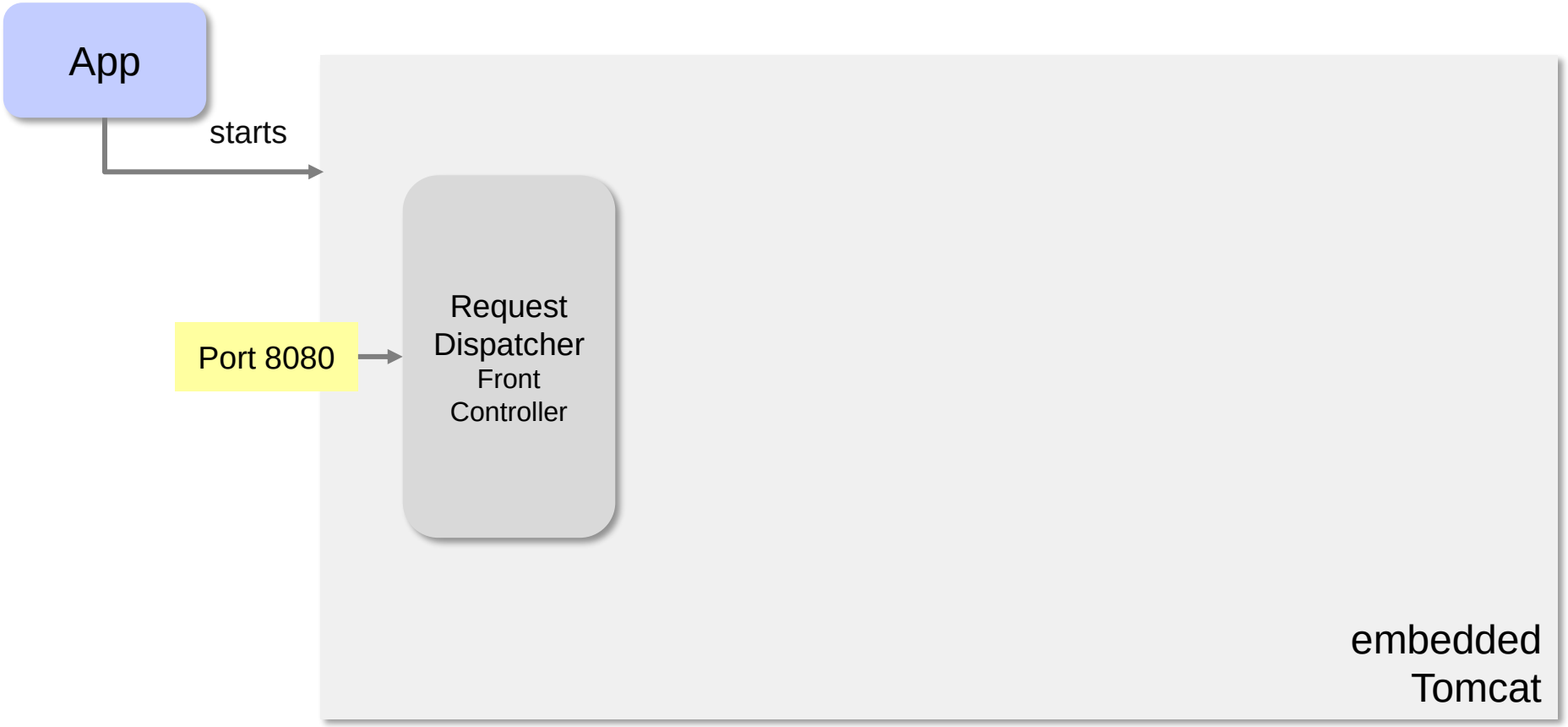
Building a Spring REST application

- Step 1: Configure app as for a Spring Boot application
- Step 2: Plan your restful URLs e.g. /bookshop/book/{isbn}
- Step 3: Implement a Controller bean
  - Use **@RequestMapping / @GetMapping** method annotations
    - to define the URI Template for the request
  - Use **@PathVariable** method parameter annotation
    - Multiple **@PathVariable** annotations can be used
    - to bind to multiple URI Template variables
  - Also **@RequestBody** indicating Spring should try and convert content of incoming message to parameter object
  - **@ResponseBody** indicates response is to be written to HTTP response
- Step 4: Setup how response body handled

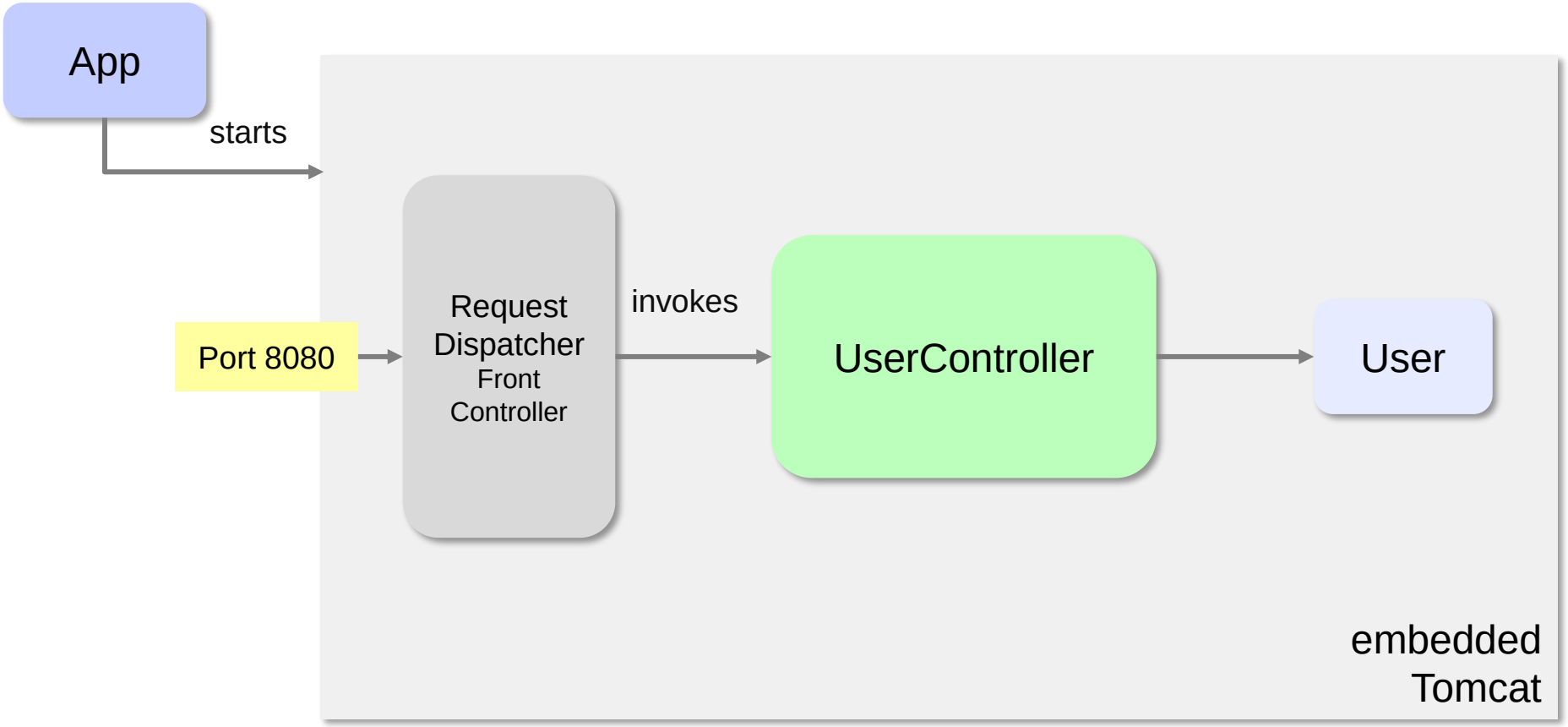
Restful Service



Restful Service



Restful Service



RESTFul Controller

```
package com.jjh.controller;

import java.util.ArrayList;
import java.util.List;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.jjh.domain.User;

@RestController
@RequestMapping("users")
public class UserController {
	private final List<User> users = new ArrayList<User>();

	public UserController() {
		System.out.println("UserController.<cons>()");
		User user = new User("Bill", 30);
		users.add(user);
		user = new User("Ben", 28);
		users.add(user);
	}
...
```

RESTFul Controller

```
    ...

        @GetMapping("user/{name}")
        public User getUser(@PathVariable String name) {
                System.out.println("UserController.getUser(" + name + ")");
                User user = new User(name, 54);
                return user;
        }

        @GetMapping("list")
        public List<User> getUsers() {
                System.out.println("UserController.getUsers()");
                return users;
        }
}
```

- @RestController
  - indicates its role within the service set up
  - it handles requests / controls requests
  - view is response data / model is business logic invoked behind controller

RESTFulController

- The URL template {name} maps to the path variable name
- The operations supports the GET Method
- A User or list of user objects are being returned
- Both methods mark the returned value as being the ResponseBody (by default)
- The Response Body is processed by Spring to convert from Java object into appropriate format
  - exact details depend on Spring configuration
  - external to controller

Support for other HTTP Methods

- Just a URL
  - @GetMapping(list) or
  - @RequestMapping(value = "list")
  - public String list(ModelMap modelMap) {
- Supporting POST
  - @PostMapping / @RequestMapping(method=RequestMethod.POST)
  - `public String create(@RequestBody String content, ModelMap modelMap) {`
- Supporting PUT
  - @PutMapping / @RequestMapping(method=RequestMethod.PUT)
  - `public String update(@RequestBody String content, ModelMap modelMap) {`
- Supporting DELETE
  - @DeleteMaping("{isbn}") / @RequestMapping(value = "{isbn}", method=RequestMethod.DELETE)
  - `public String delete(@PathVariable String isbn, ModelMap modelMap) {`

Domain Object

- Just a POJO

```
public class User {
        private String name;
        private int age;
        private String title = "Mr";

        public User() {} // Required
        public User(String name, int age) {
                this.name = name;
                this.age = age;
        }
        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public int getAge() {
                return age;
        }
        public void setAge(int age) {
                this.age = age;
        }
        public String getTitle() {
                return title;
        }
        public void setTitle(String title) {
                this.title = title;
        }}
```

Spring Application

- Has a main method to run application
  - starts up an embedded web server

```
package com.jjh.main;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.ComponentScan;

@SpringBootApplication
@ComponentScan(basePackages = "com.jjh.controller")
public class Application {
    public static void main(String[] args) {
        System.out.println("Starting the web application");
        SpringApplication.run(Application.class, args);
        System.out.println("Startup complete");
        System.out.println("http://localhost:8080/users/list");
    }
}
```

Spring Classpath

- Use Maven for all dependencies

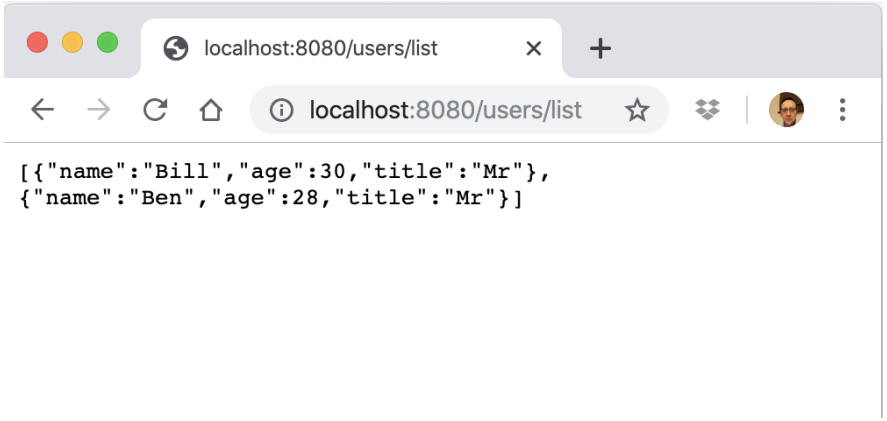
```
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-rest</artifactId>
   </dependency>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-jersey</artifactId>
   </dependency>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
   </dependency>
</dependencies>
```

Running the Service

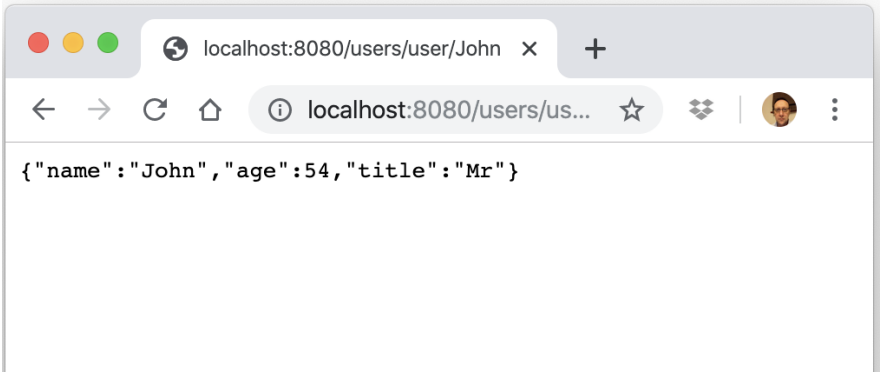
```
Starting the web application


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.4.0)

2020-11-27 17:15:42.673  INFO 14723 --- [           main] com.jjh.main.UserService
: Starting UserService using Java 14.0.1 on local with PID 14723 (/system-int)
2020-11-27 17:15:42.675  INFO 14723 --- [           main] com.jjh.main.UserService
: No active profile set, falling back to default profiles: default
2020-11-27 17:15:46.433  INFO 14723 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
: Tomcat initialized with port(s): 8080 (http)
2020-11-27 17:15:46.567  INFO 14723 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]
: Initializing Spring embedded WebApplicationContext
2020-11-27 17:15:46.567  INFO 14723 --- [           main]
w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed
in 3729 ms
2020-11-27 17:15:47.632  INFO 14723 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
: Tomcat started on port(s): 8080 (http) with context path ''
2020-11-27 17:15:47.665  INFO 14723 --- [           main] com.jjh.main.UserService
: Started UserService in 6.123 seconds (JVM running for 7.377)
Startup complete
http://localhost:8080/users/list
```

Accessing the Service

- http://localhost:8080/users/list



- http://localhost:8080/users/user/John



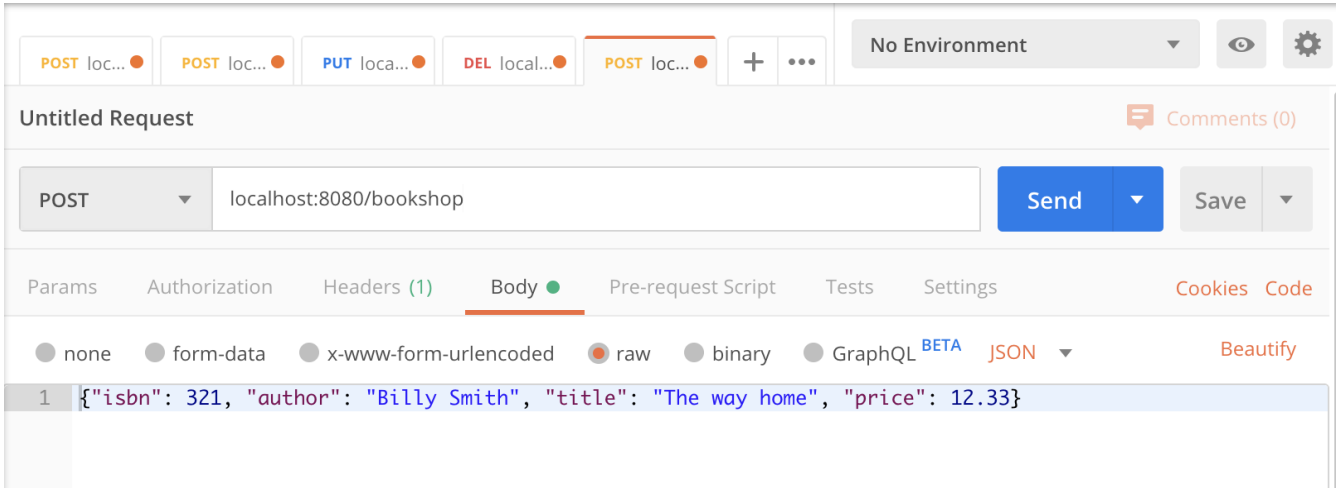
More Complete Service

```
@RestController
@RequestMapping("bookshop")
public class BookshopController {

    @Autowired
    private BookshopService bookshop;

    @GetMapping("{isbn}")
    public Book getBook(@PathVariable String isbn) {
        System.out.println("BookshopController.getBook(" + isbn + ")");
        return this.bookshop.getBook(isbn);
    }

    @GetMapping("list")
    public List<Book> getAllBooks() {
        System.out.println("BookshopController.getAllBooks()");
        return bookshop.getAllBooks();
    }

    @PostMapping
    @ResponseStatus(HttpStatus.OK)
    public void addBook(@RequestBody Book book) {
        System.out.println("BookshopController.addBook(" + book + ")");
        this.bookshop.addBook(book);
    }
```

```
    @PutMapping
    @ResponseStatus(HttpStatus.OK)
    public void updateBook(@RequestBody Book book) {
        System.out.println("BookshopController.updateBook(" + book + ")");
        this.bookshop.updateBook(book);
    }

    @DeleteMapping("{isbn}")
    @ResponseStatus(HttpStatus.OK)
    public void deleteBook(@PathVariable String isbn) {
        System.out.println("BookshopController.deleteBook(" + isbn + ")");
        bookshop.deleteBook(isbn);
    }

    @ExceptionHandler(BookNotFoundException.class)
    @ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "Book not found")
    public void updateFailure() { }


}
```

Use Postman to invoke Service

- Postman is a widely used tool for testing / accessing RESTful services
  - can specify HTTP method to use
  - can provide data for body etc.



References

- RESTFul Web Services by Leonard Richardson and Sam Ruby, O'Reilly Media, 2007.
- RESTful Java with JAX-RS 2.0 (Second Edition), Bill Burke, O'Reilly Media, 2013
- Building RESTful Web Services with JAX-RS

  https://docs.oracle.com/javaee/7/tutorial/jaxrs.htm
- "Representational State Transfer," Wikipedia article.

  https://en.wikipedia.org/wiki/Representational_state_transfer
- "RESTful Web Services: The Basics" by Alex Rodriguez, IBM developerWorks article, 2008.

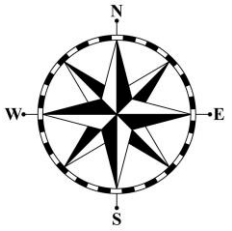
  http://www.ibm.com/developerworks/webservices/library/ws-restful/

Microservices, Service Registry and Service Lookup

Toby Dussek

Informed Academy

Plan for Session



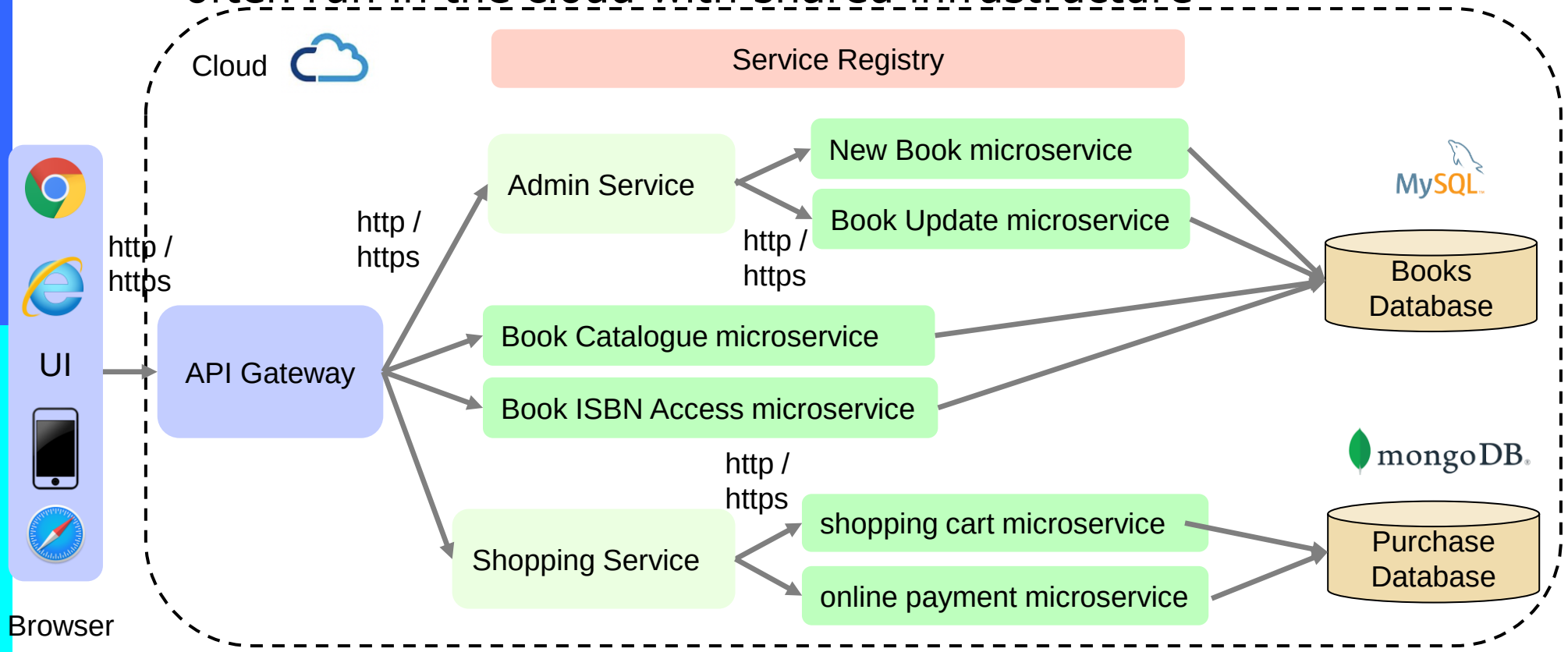
- Microservice concepts
- Microservice Architectures
- Components in a microservice Architecture
- Service Discovery
- Service Registry
- Example Services

Microservices

- Term first used at a cloud computing conference
  - by Dr. Peter Rodgers in 2005
- Google, Facebook, and Amazon have employed this approach at some level for more than ten years
- More recently, microservices have gained popularity 'cos can support more agile development, such as:
  - Incrementally adding functionality
  - Containerization of operating systems
  - Containerization of applications

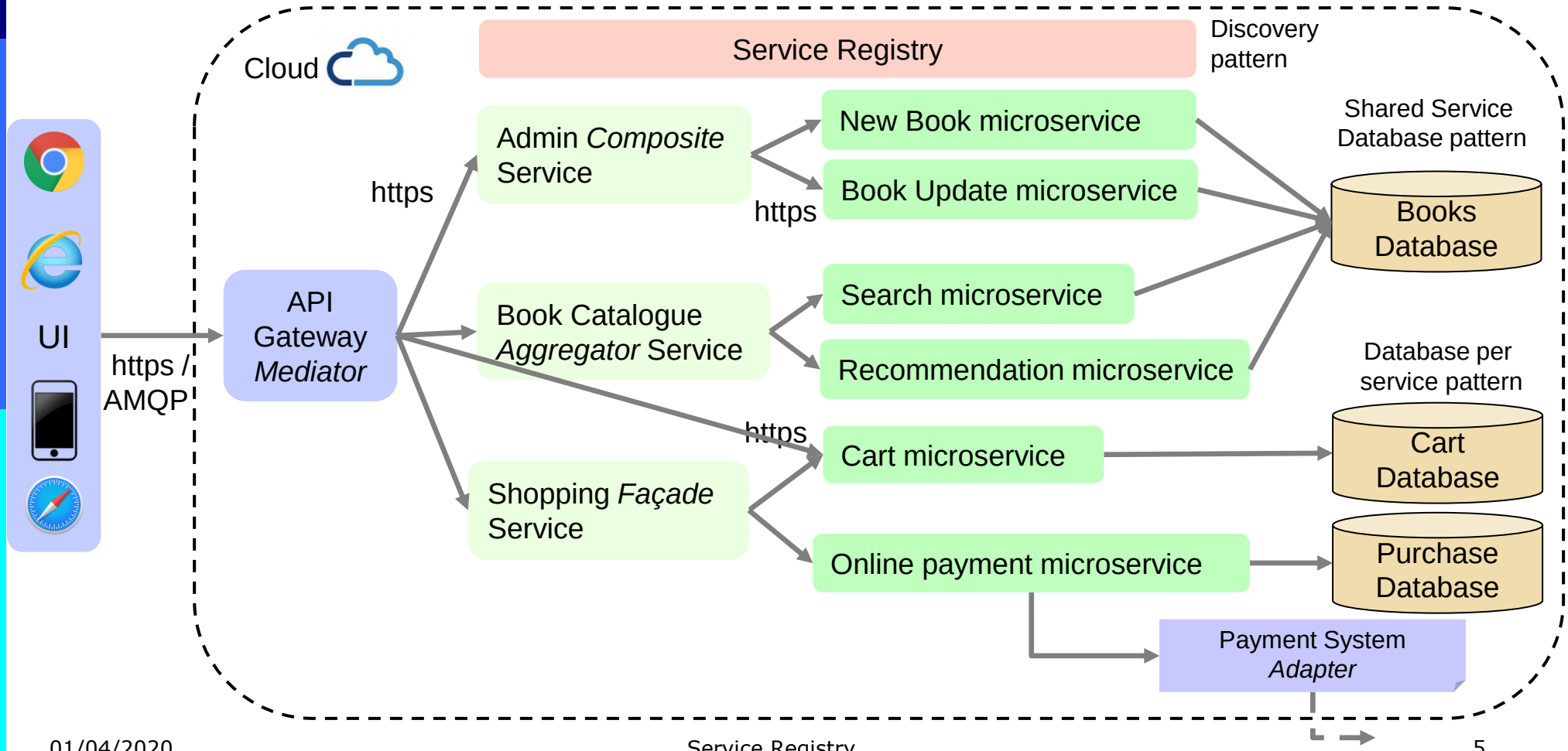
Microservices Architecture

- Break systems down into finer grained services
  - services do one thing and do it well
  - often run in the cloud with shared infrastructure



Microservice Architecture

- Typically contains multiple design patterns



Microservices - Definition

- Microservices are a type of **software architecture**
- Microservices are small, focused components built to do a single thing very well
- A microservices architecture consists of a collection of small, autonomous services.
  - Each service is self-contained and should implement a single business capability.
  - Each microservice is focused on providing a single *service*
  - More complex operations can be constructed by combining microservices together

Components in a Typical Microservices Architecture

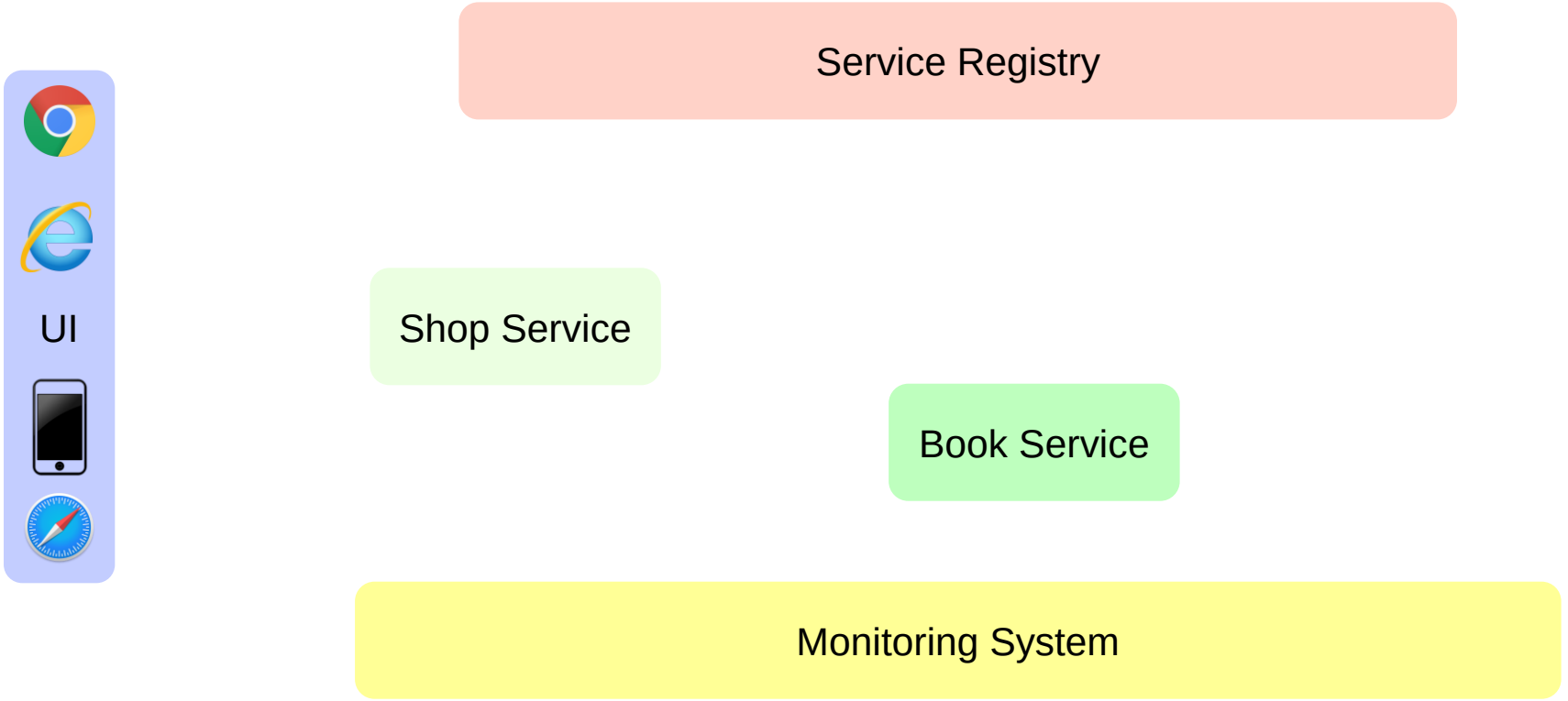
- Management
  - responsible for placing services on nodes
- Service Discovery
  - Enables service lookup to find the endpoint for a service
- API Gateway
  - entry point for clients / forwards the call to services
- Monitoring
  - tools to help monitor individual micro services
- Load Balancers
  - so that multiple instances of a service can be deployed

Service Discovery

- Make it simple for services to find other services
- Provide infrastructure to
  - handle load balancing
  - handle service registration
  - service discovery
- Ensure Service calls are location independent
  - mapping provided by service registry
  - from local name e.g. http://book-service
  - to physical address e.g. http://localhost:2222
- Provide for management and monitoring services

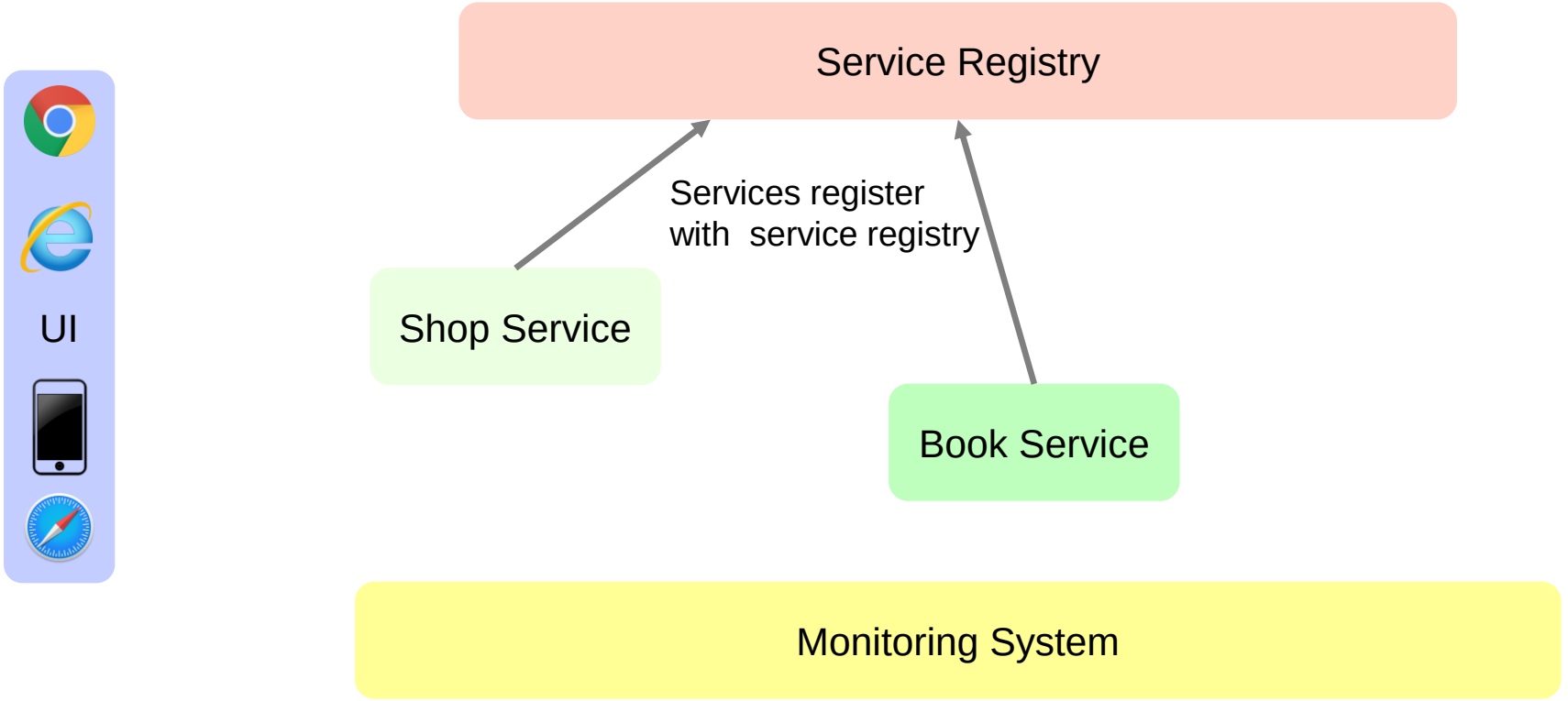
Sample Application

- Simple microservice application



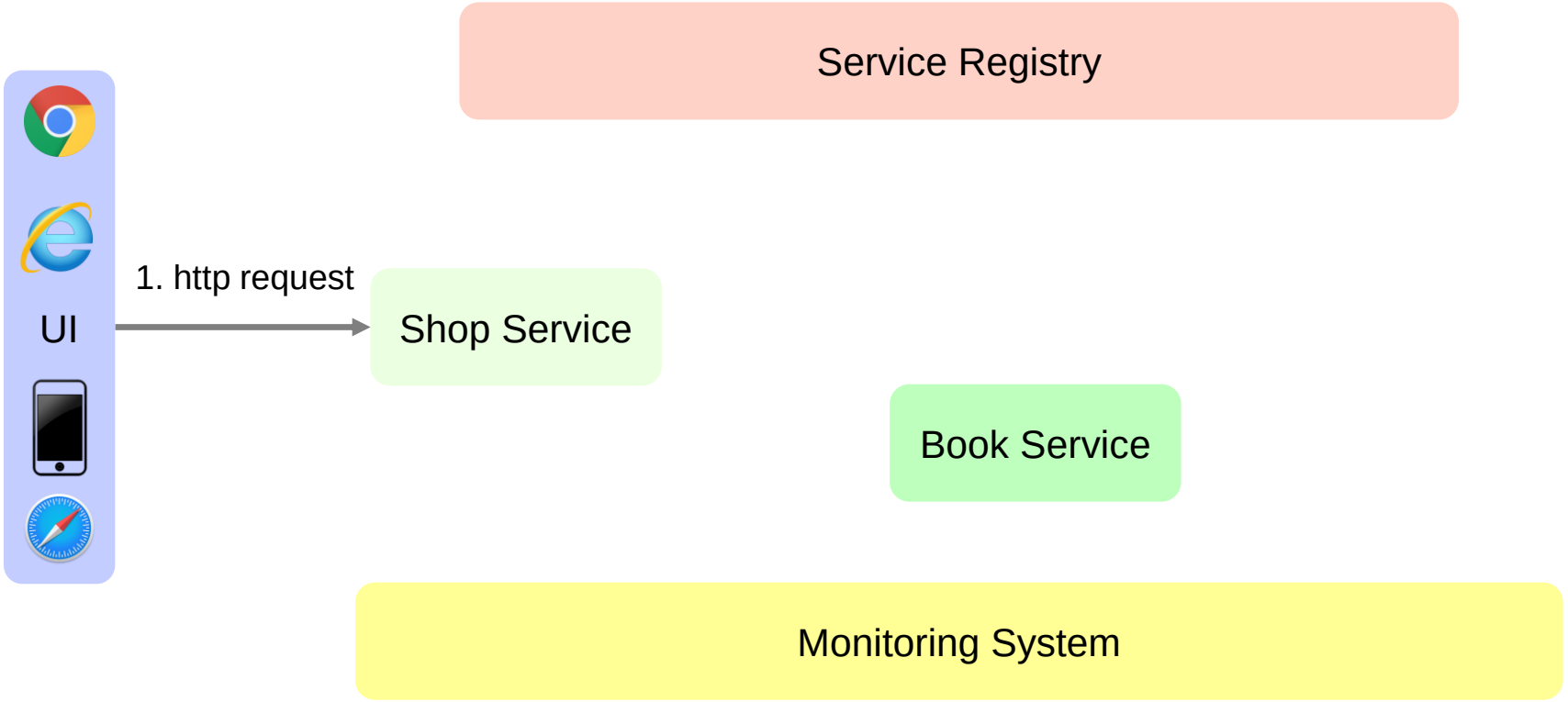
Sample Application

- Simple microservice application



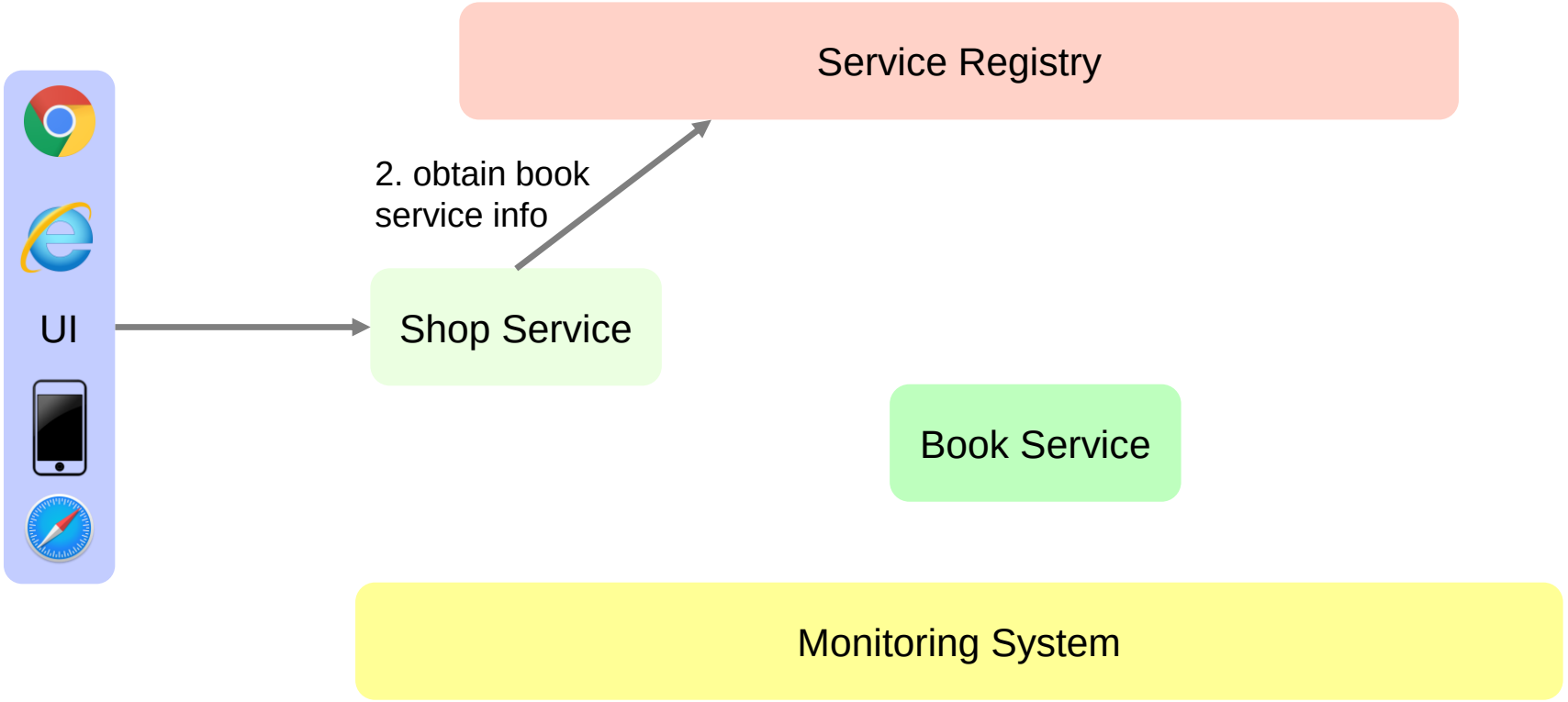
Sample Application

- Simple microservice application



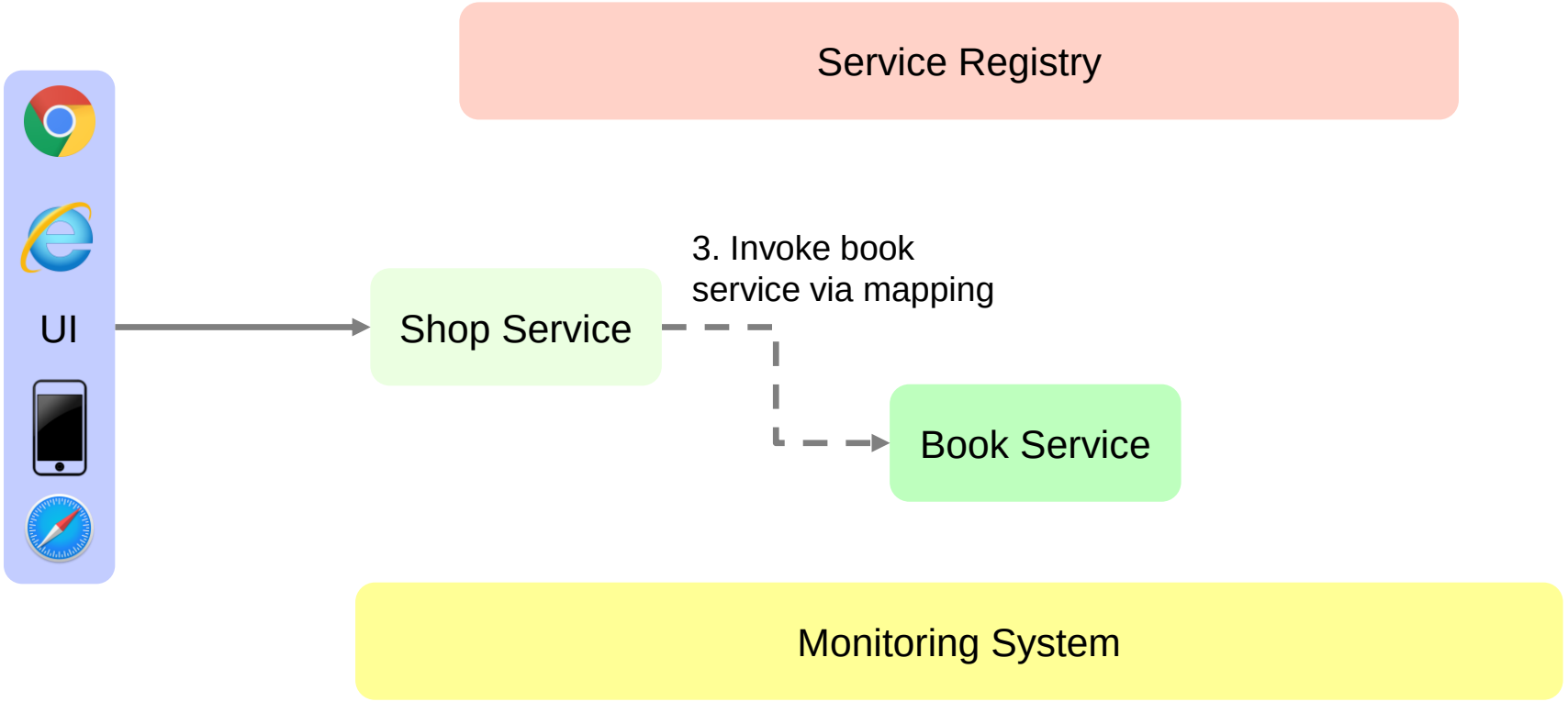
Sample Application

- Simple microservice application



Sample Application

- Simple microservice application



Service Registry

- Services need to be able to find each other
- Eureka Service Registration and Discovery
  - open source project from Netflix
  - incorporated into Spring Cloud
  - easy to use once dependencies in place
- Can use Spring Annotations to establish server
  - @EnableEurekaServer
- Config file used to configure Eureka server
  - by default in application.yml or application.properties
  - can override with
    - System.setProperty("spring.config.name", "registration-server");

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.actuate.trace.http.HttpTraceRepository;
import org.springframework.boot.actuate.trace.http.InMemoryHttpTraceRepository;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@SpringBootApplication
@EnableEurekaServer
@Configuration
public class ServiceRegistrationServer {

	public static void main(String[] args) {
		System.out.println("Starting Registration Server");
		System.setProperty("spring.config.name", "registration-server");
		SpringApplication.run(ServiceRegistrationServer.class, args);
		System.out.println("Registration Server Started");
	}

	@ConditionalOnMissingBean
	@Bean
	public HttpTraceRepository httpTraceRepository() {
		return new InMemoryHttpTraceRepository();
	}
}
```

Service Registry

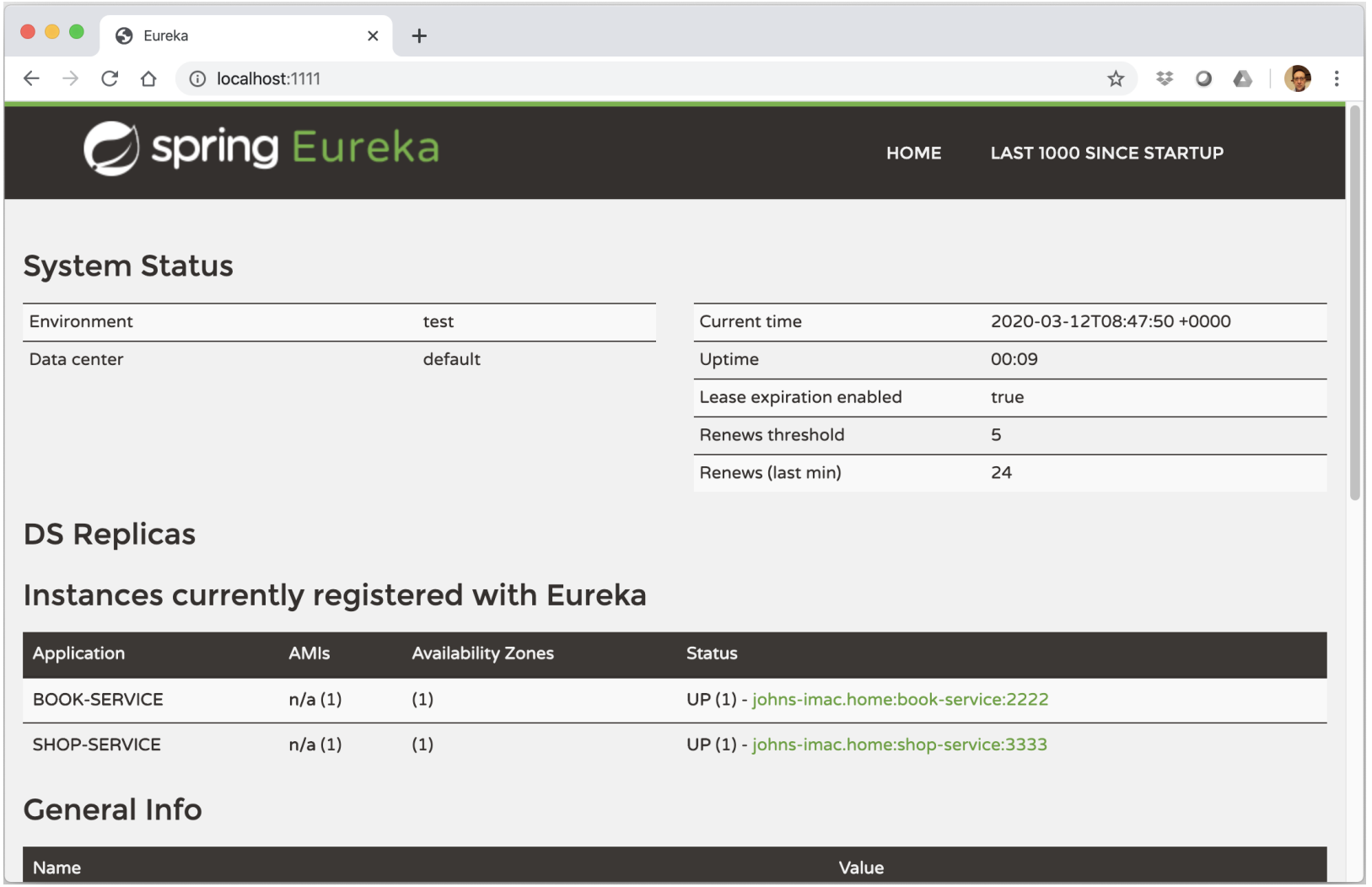
- registration-server.yml file

```
# Configure this Discovery Server
eureka:
  instance:
    hostname: localhost
  client:  # Not a client, don't register with yourself
    registerWithEureka: false
    fetchRegistry: false
  server:
    renewalPercentThreshold=0.85 # Removes a error msg

server:
  port: 1111    # HTTP (Tomcat) port
```

Running the Service Registry

```
Starting Registration Server
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.2.4.RELEASE)

2020-03-11 18:14:13.781  INFO 10693 --- [           main] c.j.m.r.ServiceRegistrationServer        : No active profile
set, falling back to default profiles: default
2020-03-11 18:14:14.282  WARN 10693 --- [           main] o.s.boot.actuate.endpoint.EndpointId     : 2020-03-11
18:14:14.719  INFO 10693 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with
port(s): 1111 (http)
2020-03-11 18:14:14.728  INFO 10693 --- [           main] o.apache.catalina.core.StandardService   : Starting service
[Tomcat]
2020-03-11 18:14:14.728  INFO 10693 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet
engine: [Apache Tomcat/9.0.30]
2020-03-11 18:14:14.799  INFO 10693 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing
Spring embedded WebApplicationContext
...
2020-03-11 18:14:16.268  INFO 10693 --- [      Thread-11] c.n.e.r.PeerAwareInstanceRegistryImpl    : Changing status to
UP
2020-03-11 18:14:16.277  INFO 10693 --- [      Thread-11] e.s.EurekaServerInitializerConfiguration : Started Eureka
Server
2020-03-11 18:14:16.292  INFO 10693 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on
port(s): 1111 (http) with context path ''
2020-03-11 18:14:16.293  INFO 10693 --- [           main] .s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to
1111
2020-03-11 18:14:16.298  INFO 10693 --- [           main] c.j.m.r.ServiceRegistrationServer        : Started
ServiceRegistrationServer in 3.03 seconds (JVM running for 3.321)
Registration Server Started
```

Accessing the Eureka Registry



Book Service

- Really a standard RESTFul service
- But registers itself with the Eureka Service Registry
  - when service application starts up
- Can use an annotation for registration
  - @EnableDiscoveryClient
- Plus configuration file
  - either property file or a YAML file

Book Service

```
...
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication
@EnableDiscoveryClient
public class BookshopService {

	public static void main(String[] args) {
		System.out.println("Starting BookshopService");

		// Tell server to look for book-server.properties or book-server.yml
		System.setProperty("spring.config.name", "book-service");

		SpringApplication.run(BookService.class, args);
		System.out.println("BookService start up completed");
	}

}
```

Book Service

```
# Spring properties
spring:
  application:
     name: book-service

# HTTP Server
server:
  port: 2222   # HTTP (Tomcat) port

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:1111/eureka/
  instance:
    leaseRenewalIntervalInSeconds: 5
```

Book Service Controller

- Nothing special here

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class BookController {

	private List<Book> books = Stream
			.of(new Book("1", "Java Today", "John", 15.55),
				new Book("2", "Spring 2.x", "Paul", 12.99))
			.collect(Collectors.toList());

	@GetMapping
	public List<Book> getBooks() {
		System.out.println("BookService.getBooks() - " + books);
		return this.books;
	}

}
```

Running the Book Service

```
Starting BookService
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.2.4.RELEASE)

2020-03-12 08:39:04.381  INFO 11585 --- [           main] com.jjh.microservices.books.BookService  : No active profile
set, falling back to default profiles: default
2020-03-12 08:39:04.961  INFO 11585 --- [           main] o.s.cloud.context.scope.GenericScope     : BeanFactory
id=7119415d-9e72-398e-a0b7-6caadaac169e
2020-03-12 08:39:05.222  INFO 11585 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized
with port(s): 2222 (http)
2020-03-12 08:39:05.229  INFO 11585 --- [           main] o.apache.catalina.core.StandardService   : Starting service
[Tomcat]
2020-03-12 08:39:05.229  INFO 11585 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet
engine: [Apache Tomcat/9.0.30]
2020-03-12 08:39:05.298  INFO 11585 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing
Spring embedded WebApplicationContext
...
2020-03-12 08:39:07.274  INFO 11585 --- [nfoReplicator-0] com.netflix.discovery.DiscoveryClient    :
DiscoveryClient_BOOK-SERVICE/johns-imac.home:book-service:2222: registering service...
2020-03-12 08:39:07.314  INFO 11585 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on
port(s): 2222 (http) with context path ''
2020-03-12 08:39:07.315  INFO 11585 --- [           main] .s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to
2222
2020-03-12 08:39:07.320  INFO 11585 --- [           main] com.jjh.microservices.books.BookService  : Started
BookService in 3.432 seconds (JVM running for 3.715)
BookService start up completed
2020-03-12 08:39:07.400  INFO 11585 --- [nfoReplicator-0] com.netflix.discovery.DiscoveryClient    :
DiscoveryClient_BOOK-SERVICE/johns-imac.home:book-service:2222 - registration status: 204
```

Shop Service

- Will need to access the BookService
- Will use the RestTemplate to do this
- Spring can handle load balancing and service discovery
- Template will be managed by Eureka
  - will work with service registry
- Use logical name of service
  - based on information used to register book service
  - registry will map to actual URL

Delegate Class used to Access BookService

```
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.web.client.RestTemplate;
import org.springframework.stereotype.Component;
import com.jjh.microservices.books.Book;

@Component
public class BookServiceDelegate {

    @Autowired
    @LoadBalanced
    private RestTemplate restTemplate;

    public List<Book> getBooks() {
        List<Book> results =
            (List<Book>)restTemplate.getForObject("http://book-service", List.class);
        return results;
    }
}
```

Shop Controller

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import com.jjh.microservices.books.Book;

@RestController
public class ShopController {

    @Autowired
    private BookServiceDelegate delegate;

    @GetMapping
    public Map<String, List<Book>> getBooks() {
        System.out.println("ShopService.getBooks()");
        Map<String, List<Book>> map = new HashMap<String, List<Book>>();
        map.put("Technical", this.delegate.getBooks());
        return map;
    }
}
```

Shop Service

```
@SpringBootApplication
@EnableDiscoveryClient
@Configuration
@ComponentScan
public class ShopService {

    public static void main(String[] args) {
        System.out.println("Starting the Shop Service");
        // Will configure using shop-service.yml
        System.setProperty("spring.config.name", "shop-service");
        SpringApplication.run(ShopService.class, args);
        System.out.println("Shop Service started");
    }

    @LoadBalanced
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

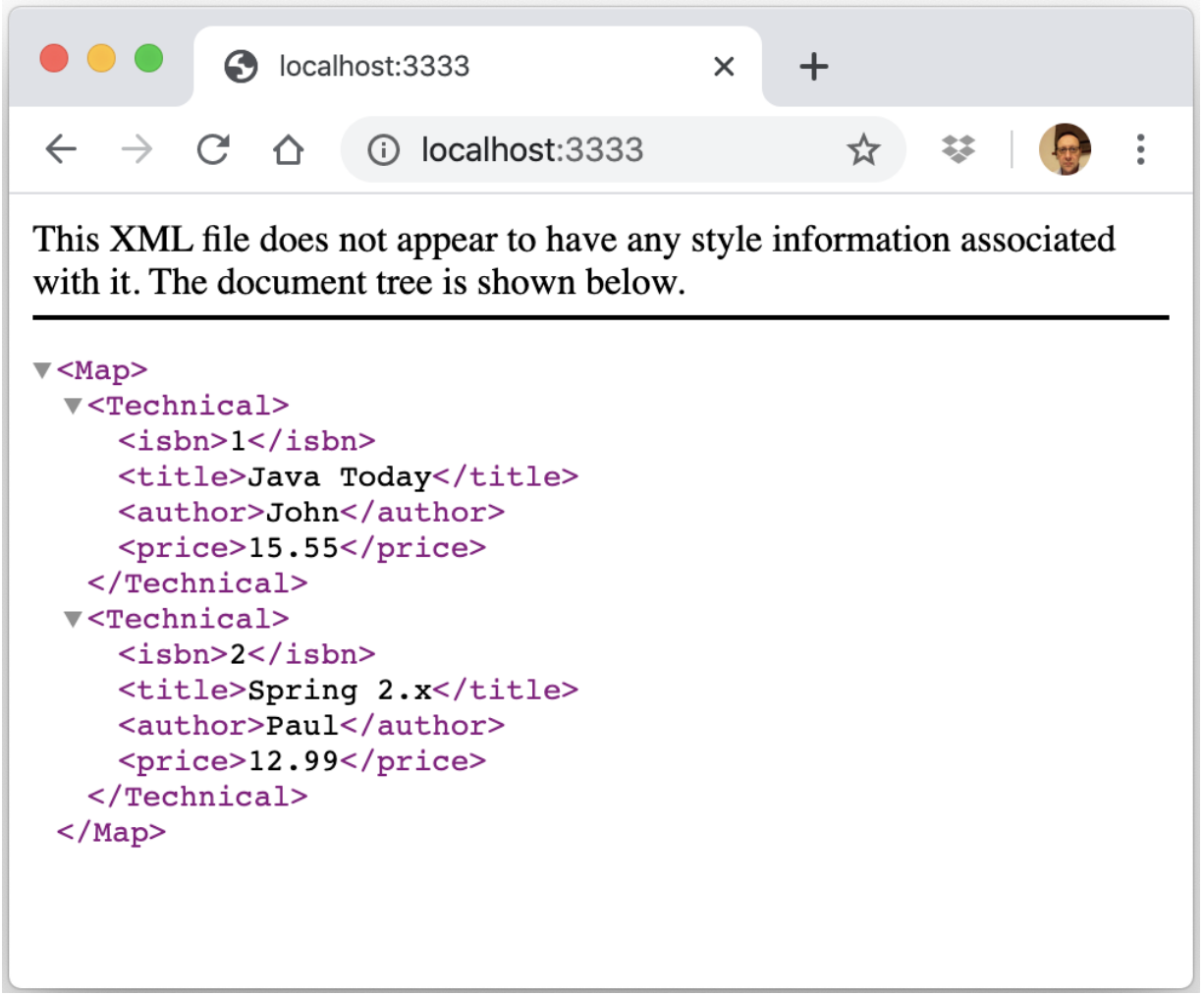
Book Service

```
# Spring properties
spring:
  application:
     name: shop-service

# HTTP Server
server.port: 3333    # HTTP (Tomcat) port

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:1111/eureka/
  instance:
    leaseRenewalIntervalInSeconds: 5
```

Running the Shop Service

```
Starting Registration Server


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.2.4.RELEASE)

2020-03-12 08:38:51.079  INFO 11584 --- [           main] c.j.m.r.ServiceRegistrationServer        : No active profile
set, falling back to default profiles: default
2020-03-12 08:38:52.048  WARN 11584 --- [           main] o.s.boot.actuate.endpoint.EndpointId     : Endpoint ID
'service-registry' contains invalid characters, please migrate to a valid format.
2020-03-12 08:38:52.281  INFO 11584 --- [           main] o.s.cloud.context.scope.GenericScope     : BeanFactory
id=0987ebd4-aa65-380b-b8e5-5d28cf7fec7e
2020-03-12 08:38:52.932  INFO 11584 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized
with port(s): 1111 (http)
...
2020-03-12 08:45:08.849  INFO 11591 --- [nfoReplicator-0] com.netflix.discovery.DiscoveryClient    :
DiscoveryClient_SHOP-SERVICE/johns-imac.home:shop-service:3333: registering service...
2020-03-12 08:45:08.878  INFO 11591 --- [nfoReplicator-0] com.netflix.discovery.DiscoveryClient    :
DiscoveryClient_SHOP-SERVICE/johns-imac.home:shop-service:3333 - registration status: 204
2020-03-12 08:45:08.882  INFO 11591 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on
port(s): 3333 (http) with context path ''
2020-03-12 08:45:08.882  INFO 11591 --- [           main] .s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to
3333
2020-03-12 08:45:08.885  INFO 11591 --- [           main] com.jjh.microservices.shop.ShopService   : Started
ShopService in 2.905 seconds (JVM running for 3.193)
Shop Service started
```

Accessing the Shop Service

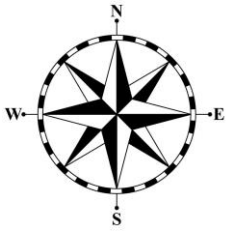


GraphQL Services

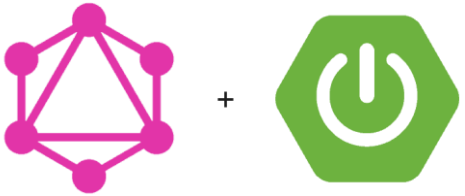
Toby Dussek

Informed Academy

Plan for Session



- Limitations of RESTful Services
- GraphQL – REST Alternative
- GraphQL Overview
- Spring Boot and GraphQL
- Creating the Schema
- POJOs represent data types in Schema
- Root Query Resolver
- Field Resolvers for Complex Types
- Book Resolver
- Spring Configuration
- Spring boot Application
- Using GraphiQL
- Mutations
- Using GraphiQL for mutations



Limitations of RESTful Services

- RESTful services work with predefined Resources
  - for example
  - http://myserver.com/books/{genre}
  - http://myserver.com/books/{id}
  - http://myserver.com/books/{author}
- But what if you want
  - all book titles written by a specific author in a specific genre
  - and the REST API designer had not thought of that
  - may need to make multiple RESTful calls
  - and handle on the client side

GraphQL – REST Alternative

- Released by Facebook in 2015
  - as an alternative to REST model
- GraphQL allows servers to publish
  - what information is available to be queried
  - in the form of data schemas
    - represented as data types and relationships between them
    - such as parent child relationships
- GraphQL allows client
  - to indicate what information it needs
  - navigation from parent to child data types
  - multiple queries in one request

GraphQL

Three key concepts

- Schema defining
  - data types and relationships
  - operations for query and mutation (update)
- QueryResolvers
  - used to handle query requests
- MutationResolvers
  - used to create / update delete data

GraphQL

- Server exposes schema
  - can be accessed by client
    - e.g. GraphiQL client
- Schema of data types and relationships
- Each data type
  - has 1 or more fields
  - take 1 or more arguments
  - return a specific type
  - may be non-nullable (indicated by !)

```
type Book {
    isbn: ID!
    title: String!
    category: String
    author: Author
}

type Author {
    id: ID!
    name: String!
    books: [Book]!
}

type Query {
    books(count: Int): [Book]!
}

type Mutation {
    writeBook(isbn: String!,
                  title: String!,
                  category: String,
                  author: String!) : Book!
}
```

GraphQL

- Queries used to specify client data requirements
- Query types defined in schema

```
type Query {
    books(count: Int): [Book]!
}
```

  - Indicates that can query for a set of books
  - will return a collection of Books
- Can specify information to retrieve
  - e.g. isbn, title and author details
  - for first 2 books in bookstore

```
query {
  books(count: 2) {
    isbn
    title
    author {
      name
    }
  }
}
```

GraphQL

- Mutations used to create / update / delete data

```
type Mutation {
    writeBook(isbn: String!,
                    title: String!,
                    category: String,
                    author: String!) : Book!
}
```

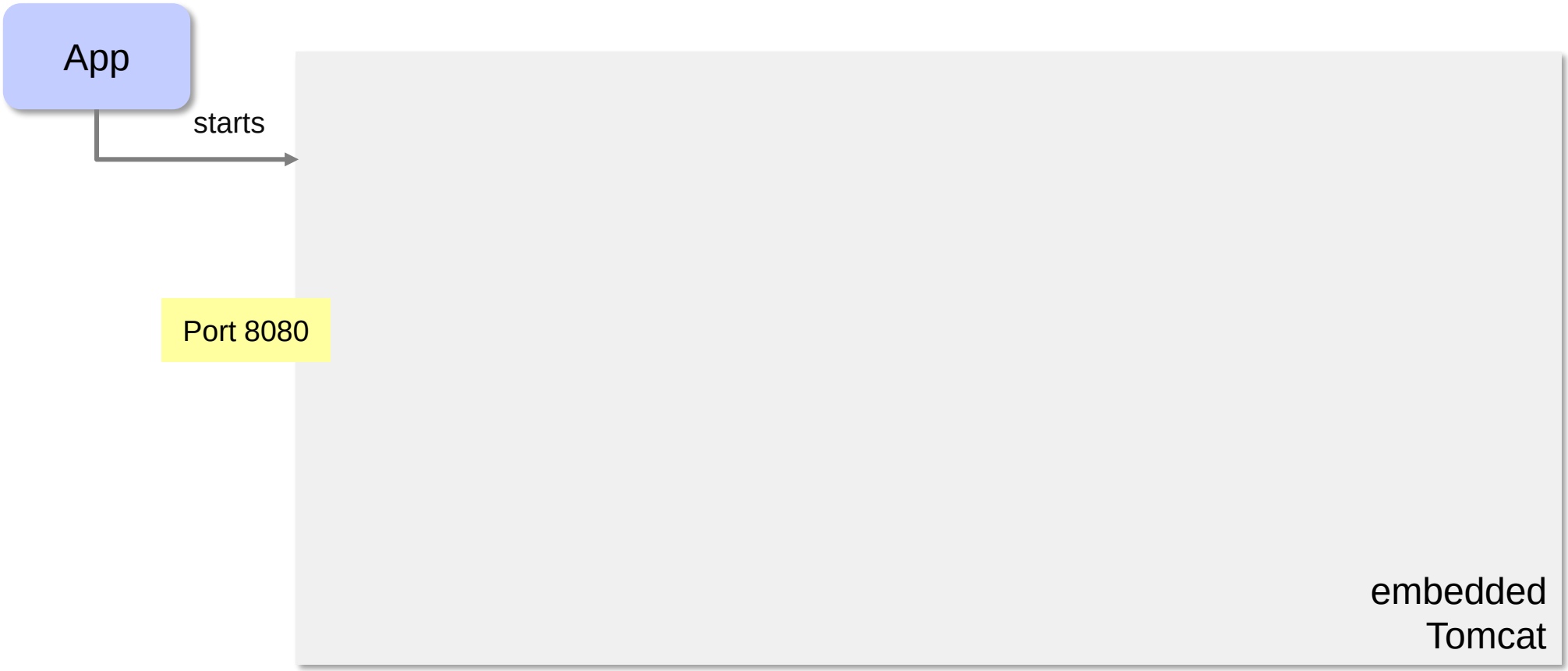
- Indicates that the writeBook function can be used to add a new book
  - takes four parameters
  - three of which are not nullable

Spring Boot and GraphQL

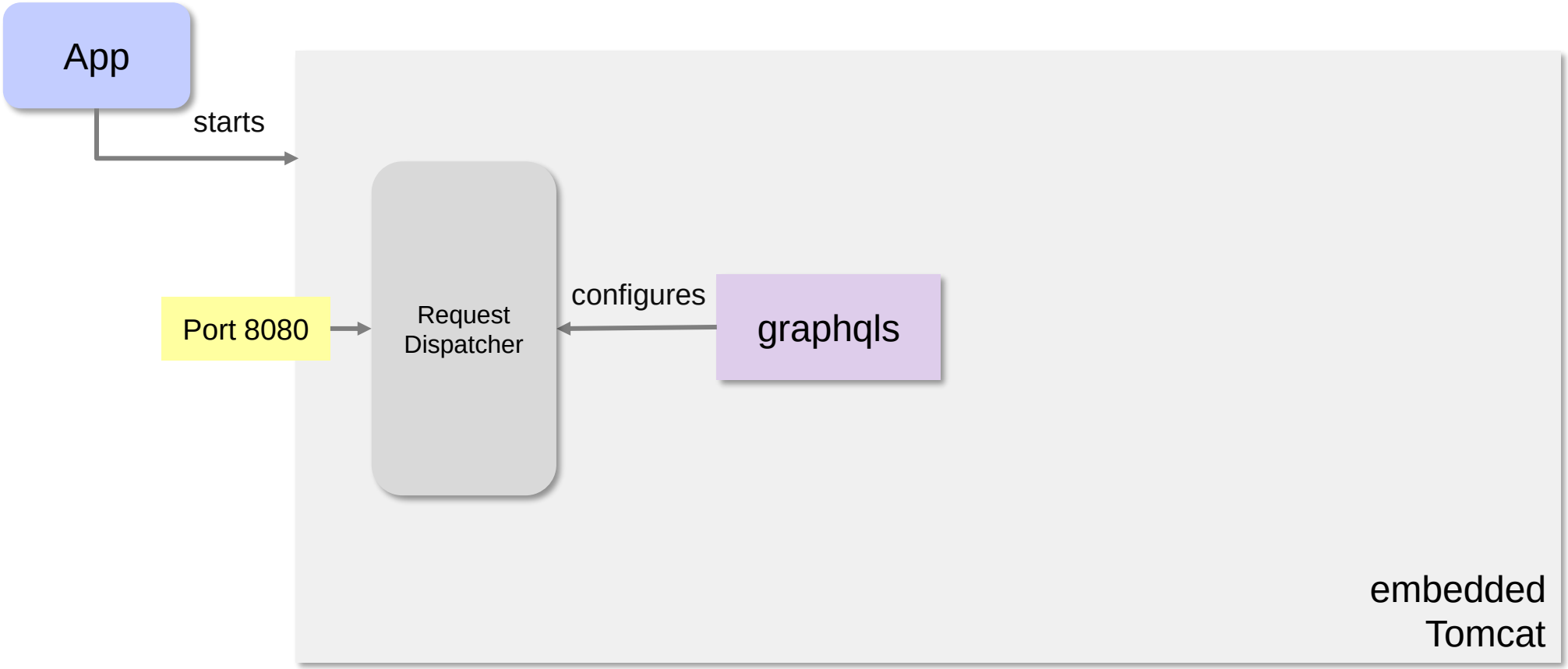
- Set of dependencies
  - build on GraphQL Java tooling

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
   <groupId>com.graphql-java-kickstart</groupId>
   <artifactId>graphql-spring-boot-starter</artifactId>
   <version>7.0.1</version>
</dependency>
<dependency>
   <groupId>com.graphql-java-kickstart</groupId>
   <artifactId>graphiql-spring-boot-starter</artifactId>
   <version>7.0.1</version>
</dependency>
<dependency>
   <groupId>com.graphql-java-kickstart</groupId>
   <artifactId>graphql-java-tools</artifactId>
   <version>6.0.2</version>
</dependency>
```

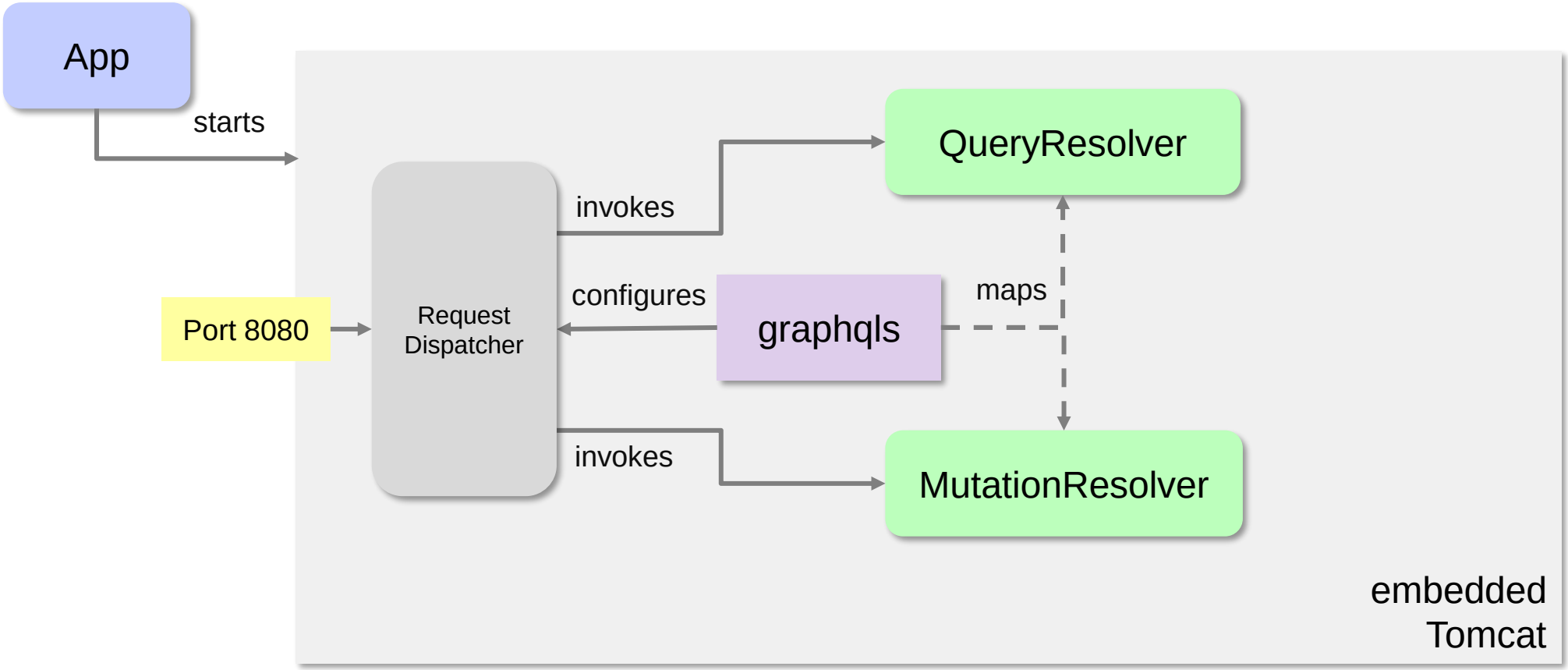
GraphQL Application



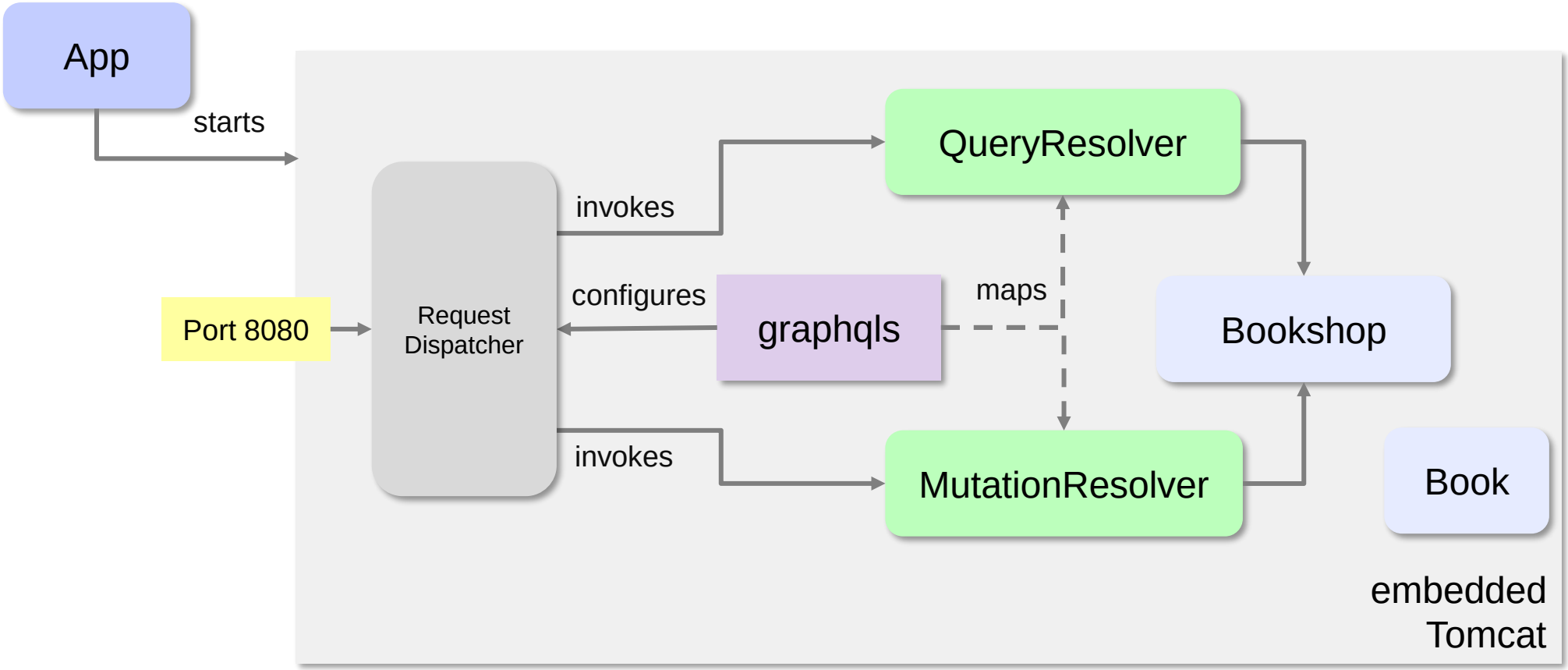
GraphQL Application



GraphQL Application

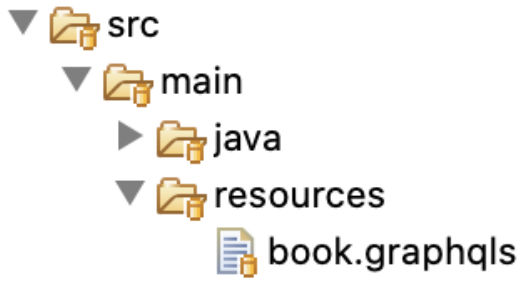


GraphQL Application



Creating the Schema

- Add book.graphqls file to resources



- Application will parse this file
  - and look for classes that match the server code requirements implied by schema

POJOs represent data types in schema

- Simple POJOS for Book and Author

```
public class Book {
        private String isbn;
        private String title;
        private String category;
        private String authorId;

        public Book() {}
        public Book(String isbn, String title, String category, String authorId) {
                this.isbn = isbn;
                this.title = title;
                this.category = category;
                this.authorId = authorId;
        }

        public String getIsbn() {
                return isbn;
        }

        public void setIsbn(String id) {
                this.isbn = id;
        }

        // ...
}
```

Root Query Resolver

- Need a bean that will handle the query

```
type Query {
    books(count: Int): [Book]!
}
```

- Bean must implement GraphQLQueryResolver
- Provide an implementation for the books query
- Name of method must be one of following, in this order:
  - <field>
  - is<field> – only if the field is of type *Boolean*
  - get<field>

Root Query Resolver

- Implements getBooks method
  - must take params
  - return the type
  - as indicated in schema

```
package com.jjh.spring.graphql;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import graphql.kickstart.tools.GraphQLQueryResolver;

@Component
public class BooksQueryResolver implements GraphQLQueryResolver {
	private Bookshop bookDao;

	@Autowired
	public BooksQueryResolver(Bookshop bookDao) {
		this.bookDao = bookDao;
	}

	public List<Book> getBooks(int count) {
		System.out.println("Query.getBooks(" + count + ")");
		return bookDao.getBooks(count);
	}
}
```

Field Resolvers for Complex Types

- Need to handle relationship between types
  - e.g. between Book and Author
    - Books hold the id of the Author
  - important as client can indicate they want parent & child
- This can be done using a Field Resolver
  - handles resolving a field value to an actual object
- Field Resolver is a bean
  - with same name as the data bean,
  - with the suffix *Resolver*,
  - and that implements the *GraphQLResolver* interface

AuthorForBookResolver

- Used to retrieve the author for a given book – not a top level query

```
@Component
public class AuthorForBookResolver implements GraphQLResolver<Book> {
	private AuthorDAO authorDao;

	@Autowired
	public AuthorForBookResolver(AuthorDAO authorDao) {
		this.authorDao = authorDao;
	}

	public Optional<Author> getAuthor(Book book) {
		System.out.println("BookQueryResolver.getAuthor(" + book + ")");
		return authorDao.getAuthor(book.getAuthorId());
	}
}
```

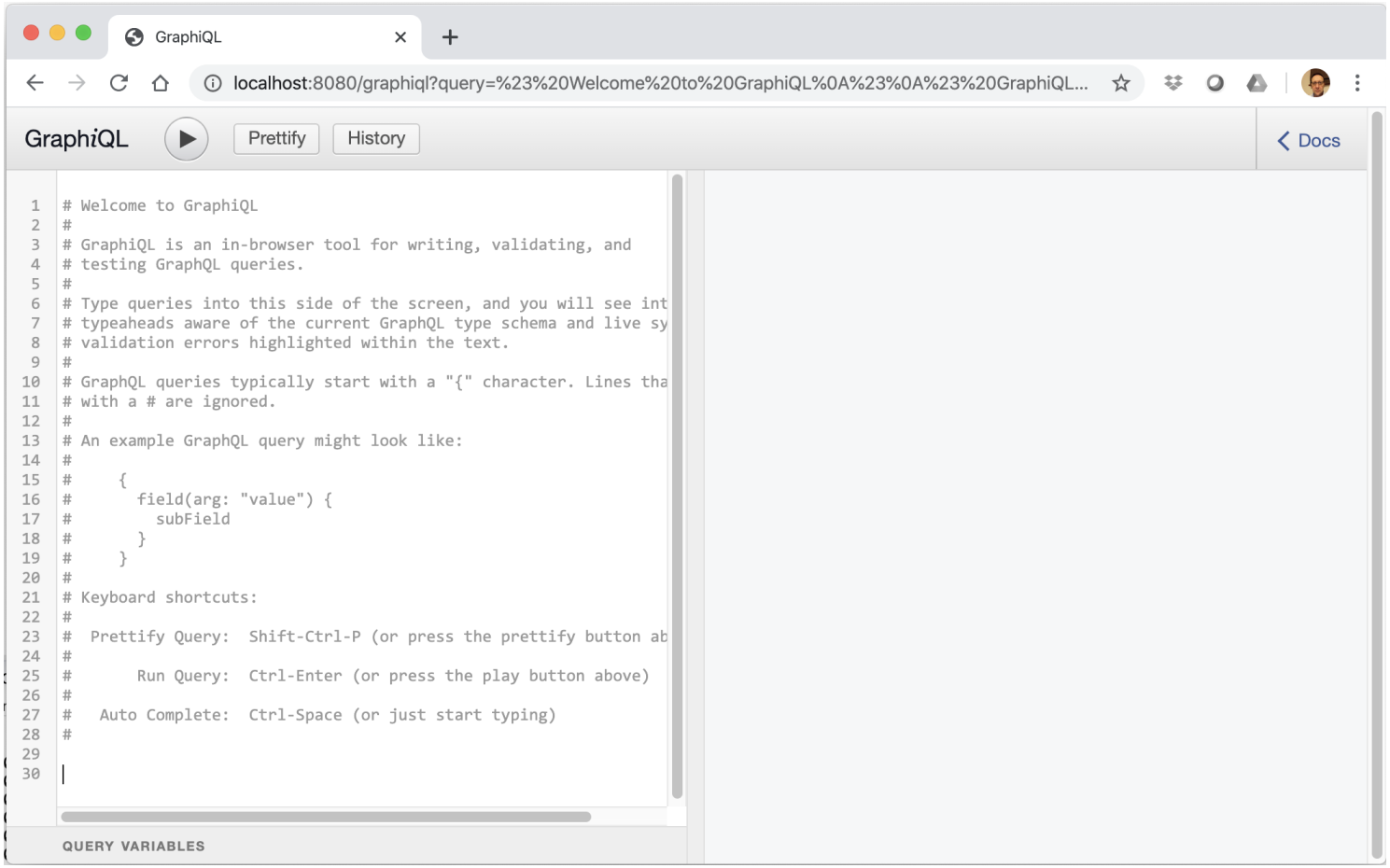
Spring Boot Application

- Standard Spring Boot Application

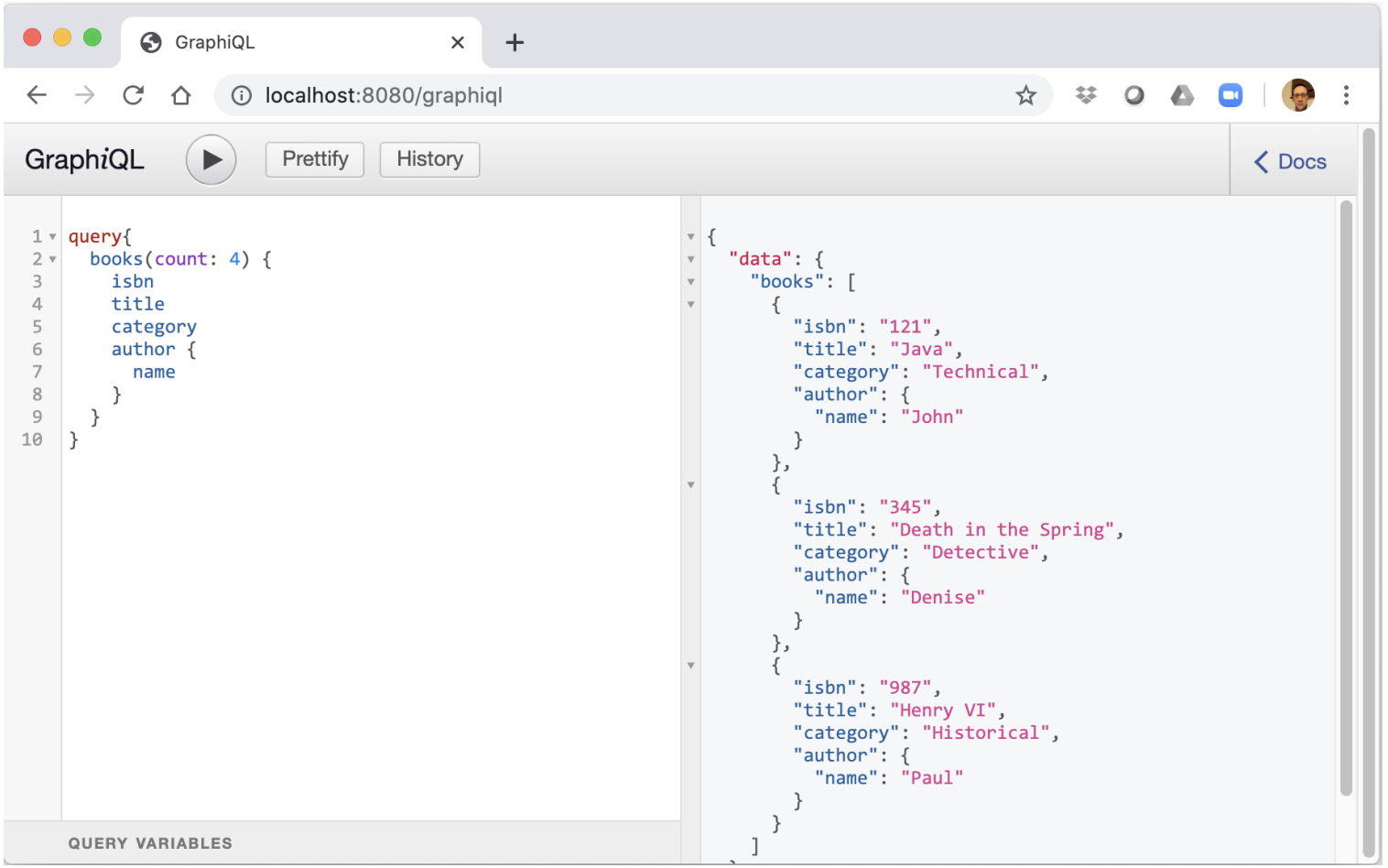
```
package com.jjh.spring.graphql;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;

@SpringBootApplication
public class BookshopGQLApp extends SpringBootServletInitializer {

	public static void main(String[] args)  {
		System.out.println("Starting App setup");
		SpringApplication.run(BookshopGQLApp.class, args);
		System.out.println("Setup finished");
	}
}
```

Using GraphiQL

http://localhost:8080/graphiql

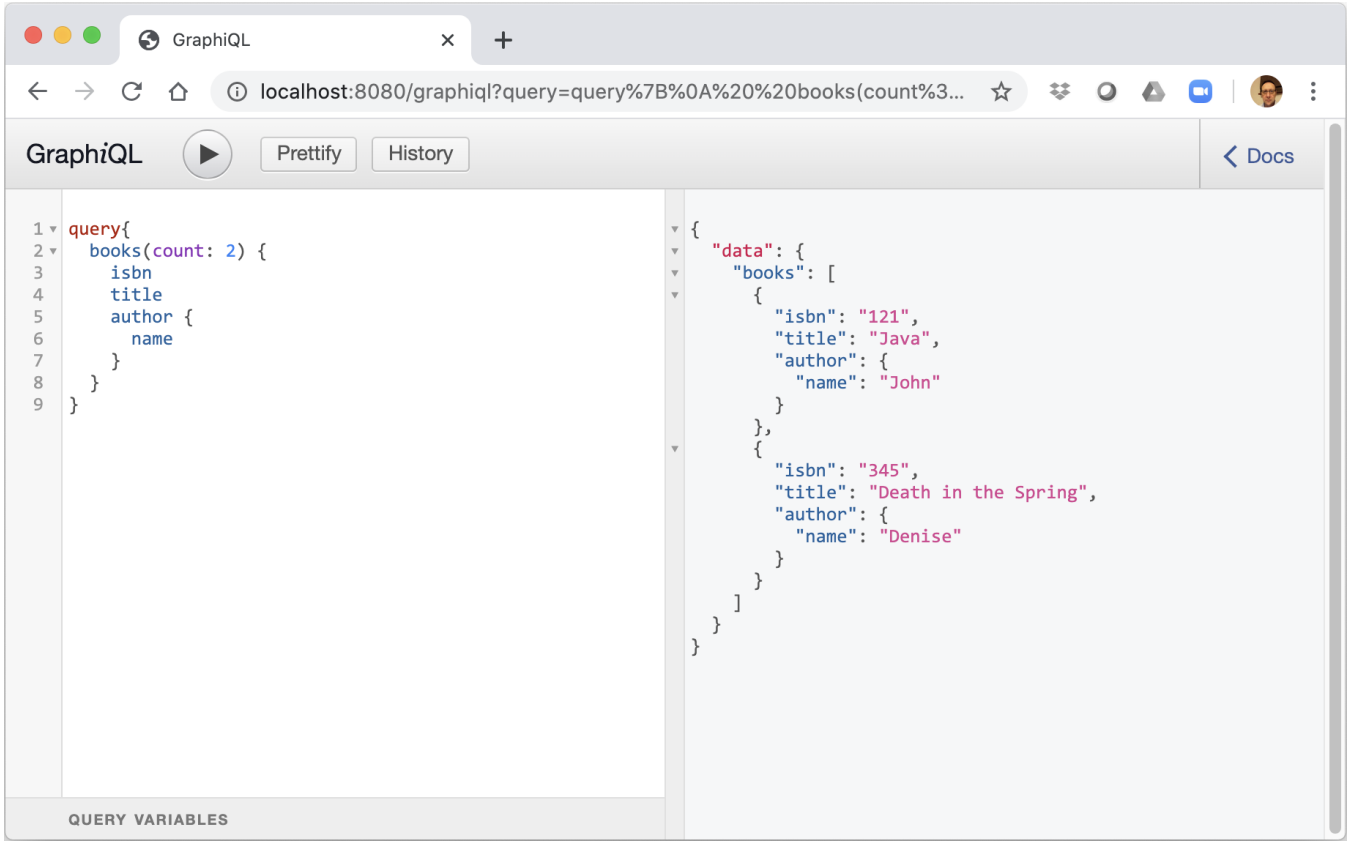


Using GraphiQL



Using GraphiQL

- Can modify query request to indicate information required



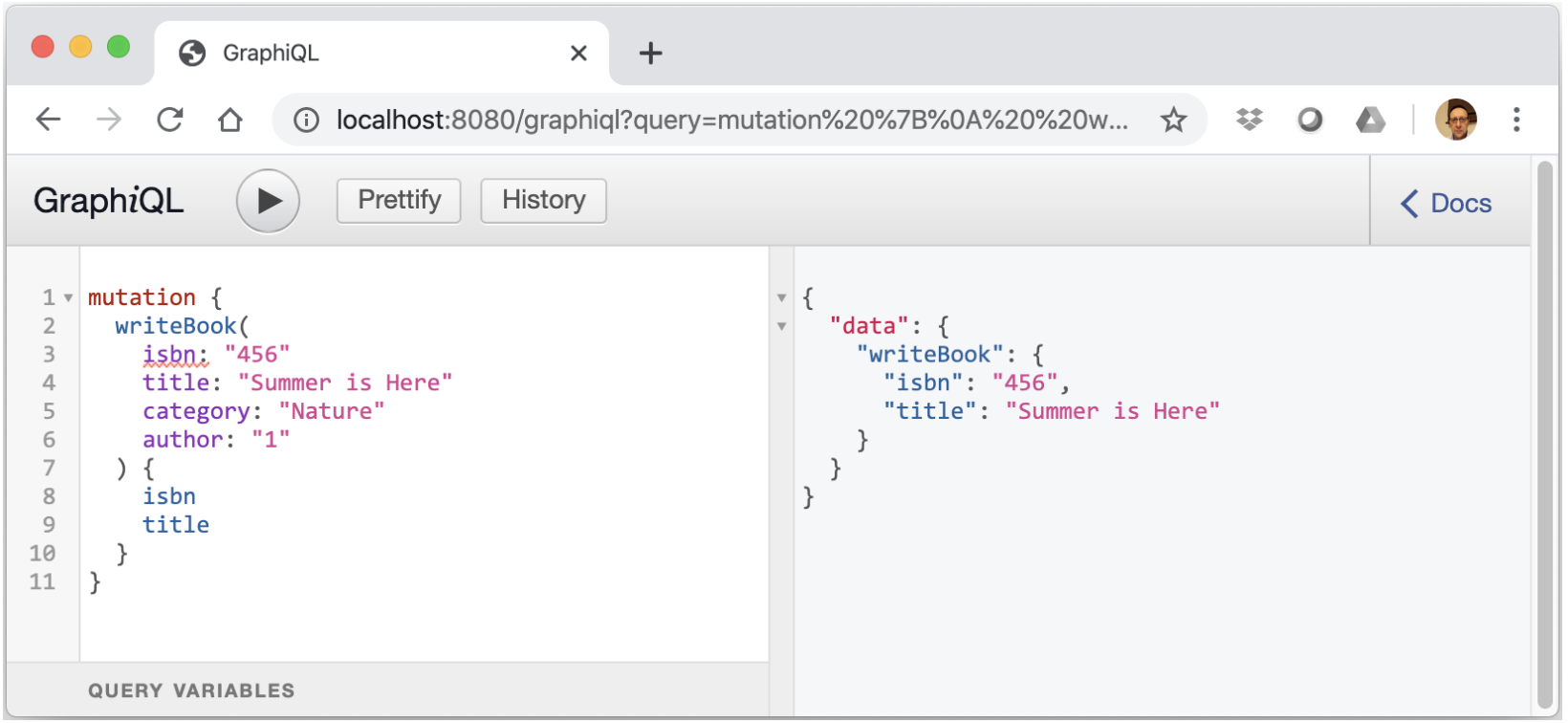
Mutations

- Mutation classes implement *GraphQLMutationResolver*

```
import graphql.kickstart.tools.GraphQLMutationResolver;

@Component
public class BookMutationResolver implements GraphQLMutationResolver {
    private Bookshop bookDao;

    @Autowired
    public BookMutationResolver(Bookshop bookDao) {
        this.bookDao = bookDao;
    }

    public Book writeBook(String isbn, String title, String category, String author) {
        Book book = new Book(isbn, title, category, author);
        bookDao.saveBook(book);
        return book;
    }
}
```

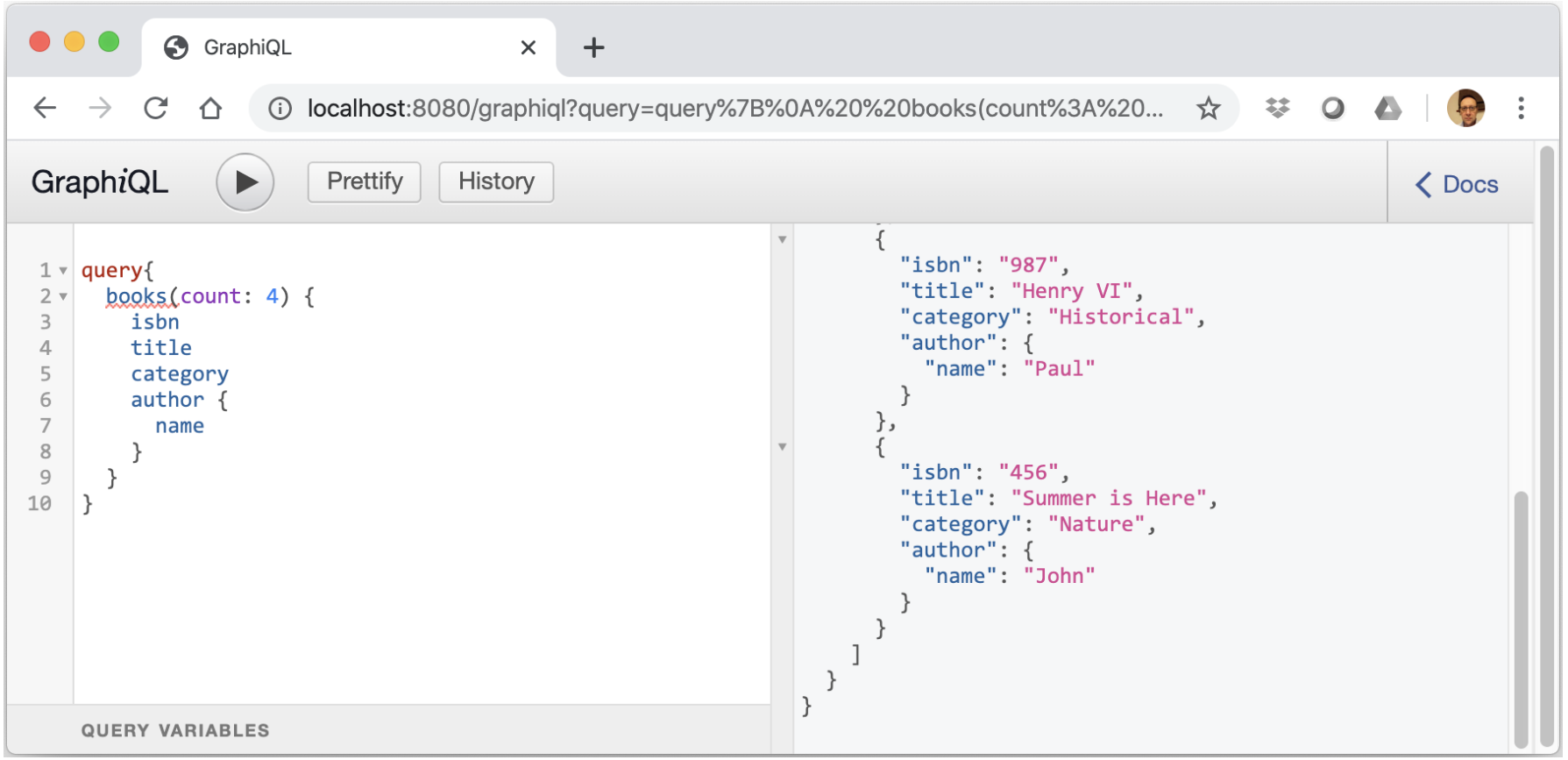
Using GraphiQL for mutation

- To Add a new book



Using GraphiQL for mutation

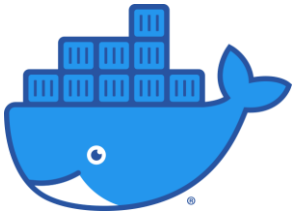
- New book now added to list



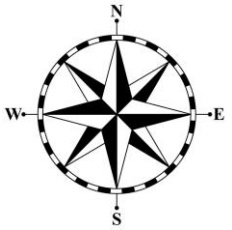
Spring Boot and Docker

Toby Dussek

Informed Academy



Plan for Session



- Motivation for Containers
- Virtual Machines V Containers
- Introduction to Docker
- Container Deployment
- Working with Docker
- Checking the Installation
- Running Spring Boot Service
- Accessing Service
- Dockerizing Spring boot App
- Add the Dockerfile
- Build the Docker Image
- Push Docker Image to Docker Hub

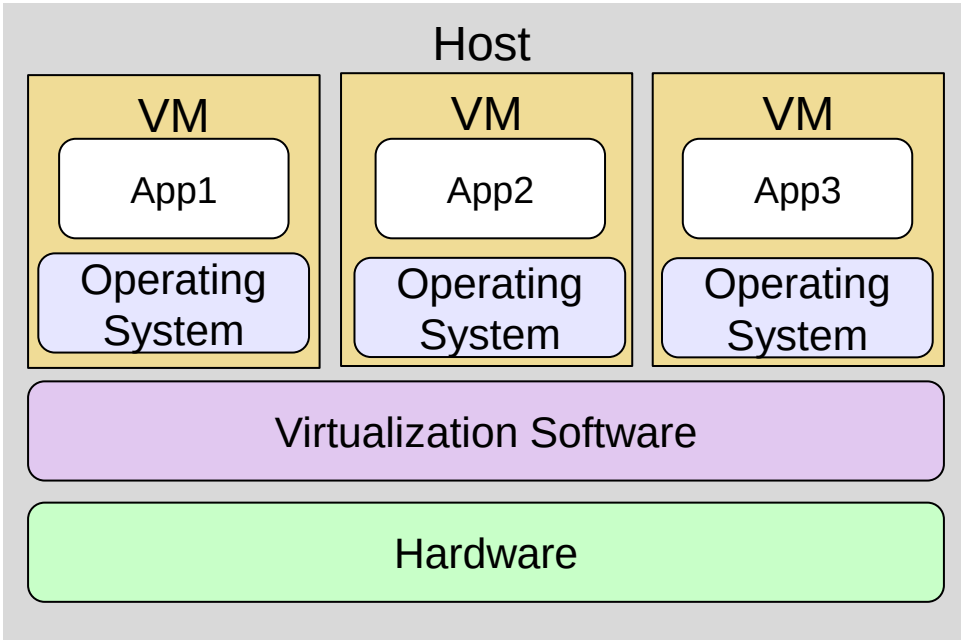
Motivation for Containers

- Have you ever encountered these issues?
  - It runs on my machine
  - You haven't got the JDK only the JRE
  - It's the wrong version of Java
  - You need Spring Boot 2.x not 1.x
  - But it only works with Tomcat and Jersey
  - Are all the environment variables set?
- Then you know why we need a container
  - so that we can create an *image* of what is needed
  - rather than a collection of files, configs and runtimes

But Virtual Machines are Expensive

- Could use VM technology to create Virtual Machines
  - each VM configured to provide right environment for an app

Apps on VMs

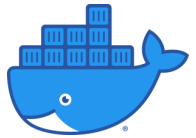


  - but this is a heavy weight solution particularly if the apps are microservices

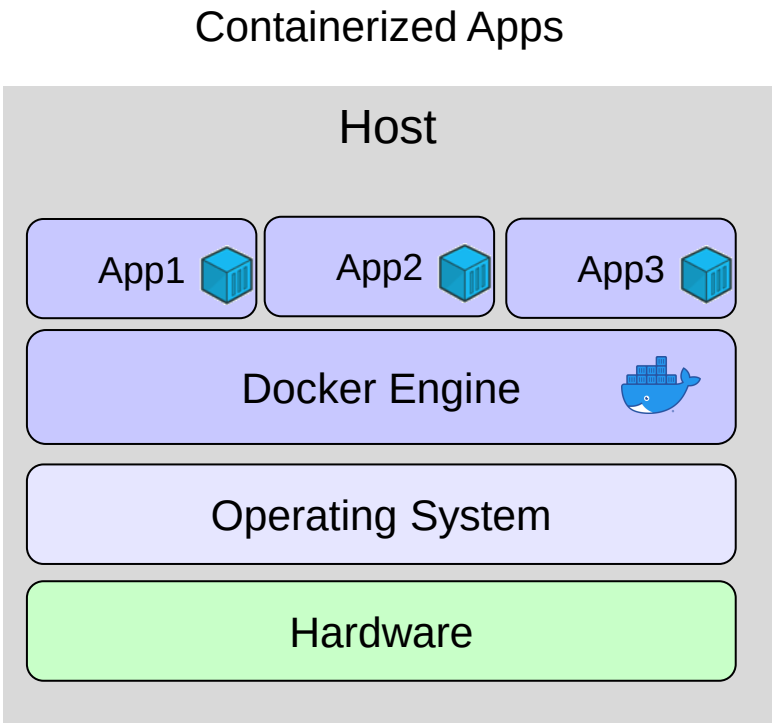
Containers to the Rescue

- Docker uses docker images
  - lightweight, standalone, executable packages
  - that includes everything needed to run app inc. classes, runtime, libraries, config etc.
  - images run within a container
  - images can be hosted centrally
  - and easily shared so that they can be run anywhere
  - whenever required
  - just using the docker command

Docker



- Each container runs an image which is isolated from every other container



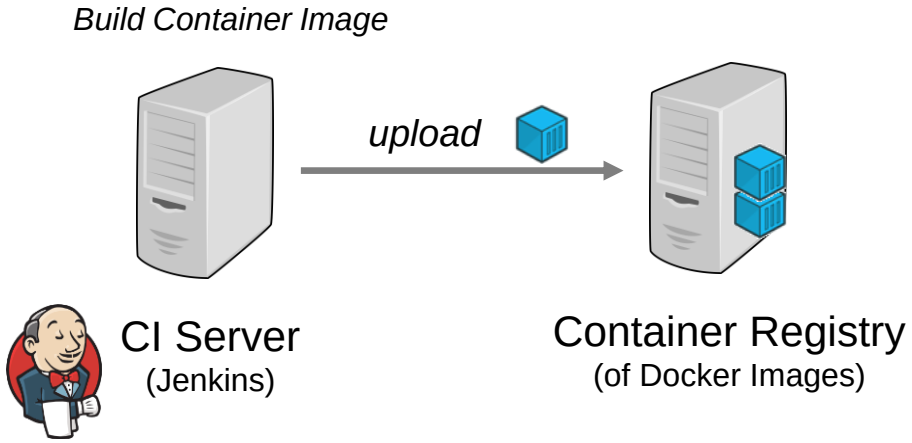
Container Deployment

Build Container Image

CI Server
(Jenkins)

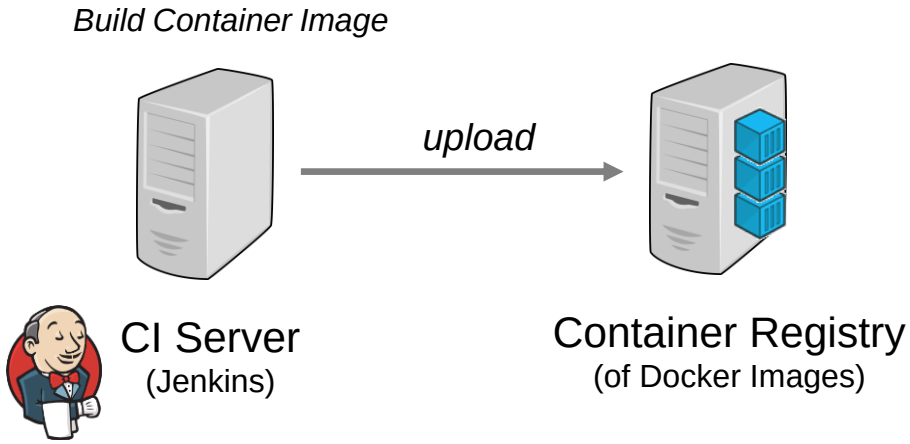
- Jenkins builds Container Images

Container Deployment



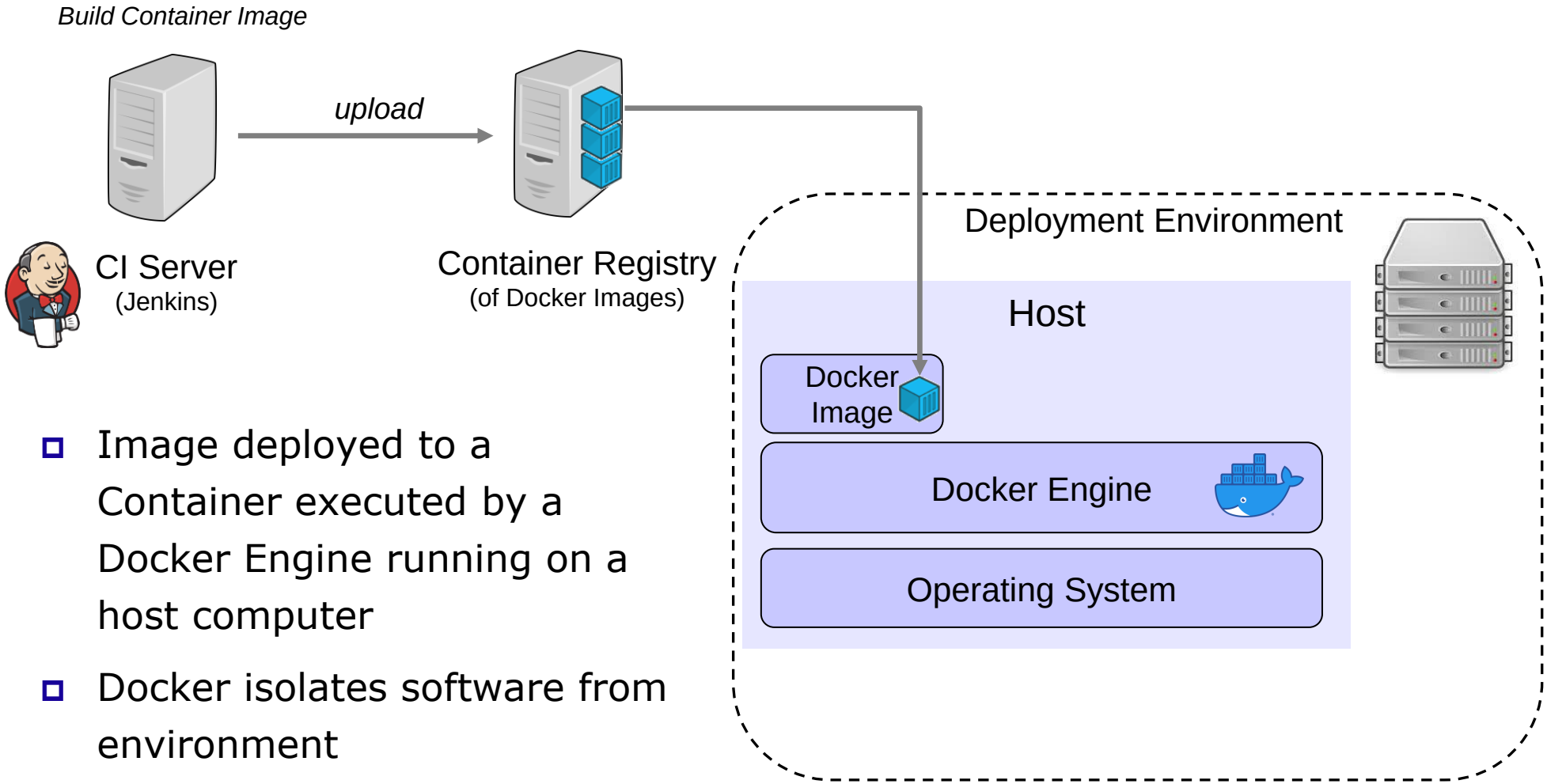
- Container images uploaded to Container Registry

Container Deployment



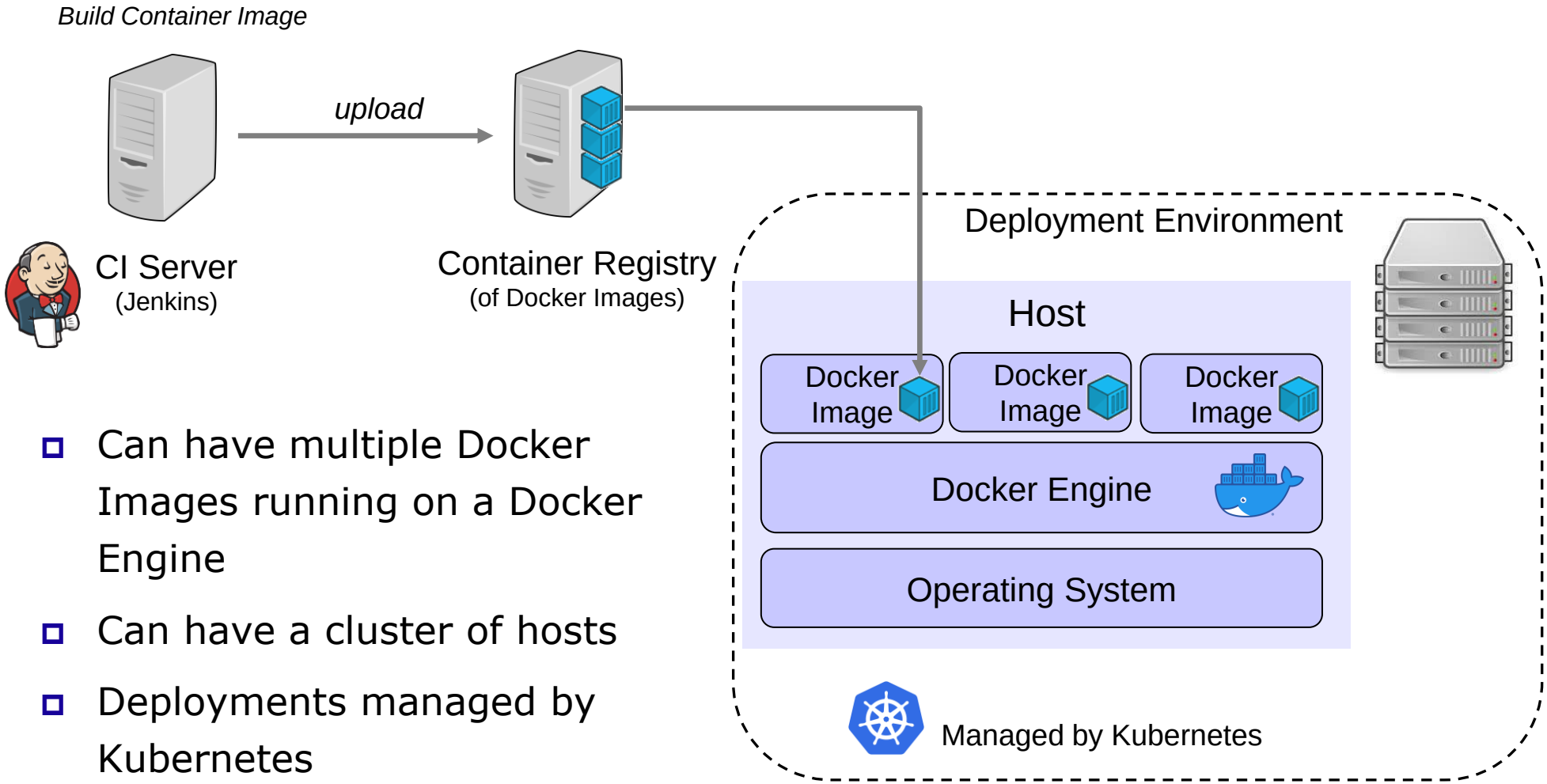
- Each image has everything needed to run app
  - code, runtime, libraries, settings, pictures, etc.
  - but not the OS so lighter weight than a Virtual Machine (VM)

Container Deployment



- Image deployed to a Container executed by a Docker Engine running on a host computer
- Docker isolates software from environment

Container Deployment



- Can have multiple Docker Images running on a Docker Engine
- Can have a cluster of hosts
- Deployments managed by Kubernetes

Working with Docker

- Will need the Docker Desktop (for Windows or Mac)
  - can download from https://hub.docker.com

  - includes the Docker Engine, the Docker CLI client, Docker Compose, Kubernetes and Credential Helper

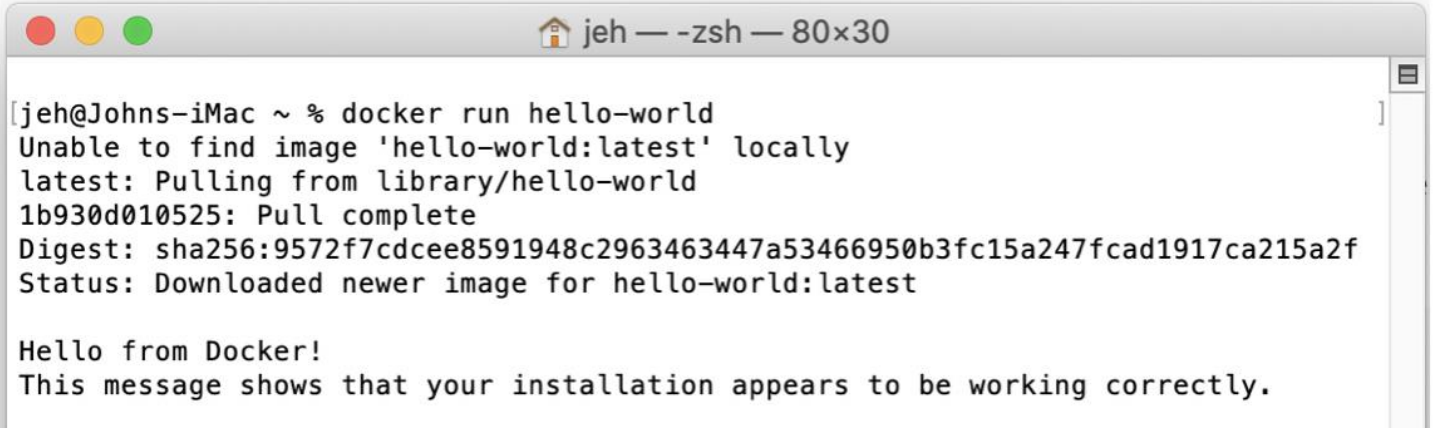
Checking the Installation

- Can use the docker CLI to check the version

```
docker --version
```

- Can run a simple Hello world app using Docker

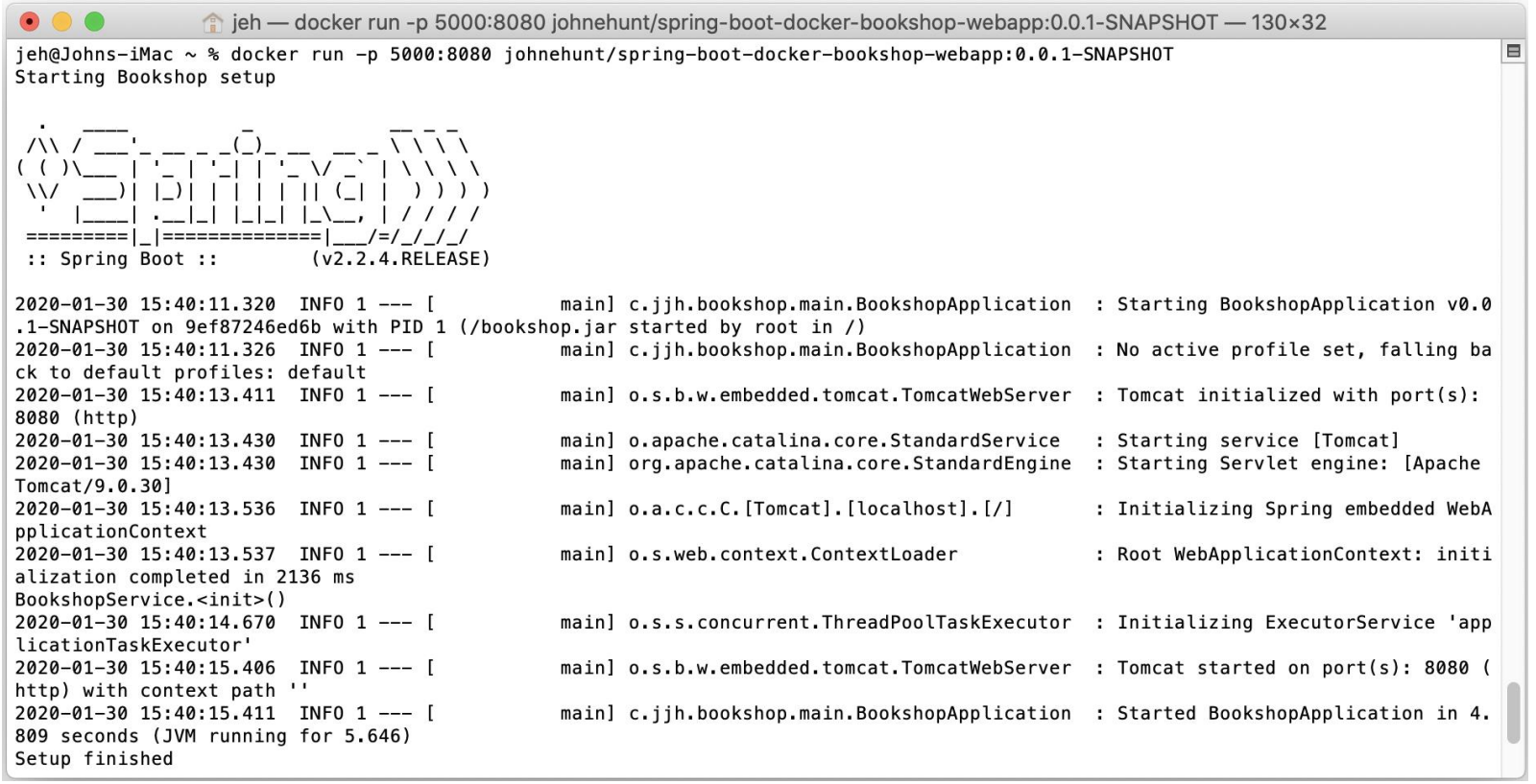
```
docker run hello-world
```



Running Spring Boot Service

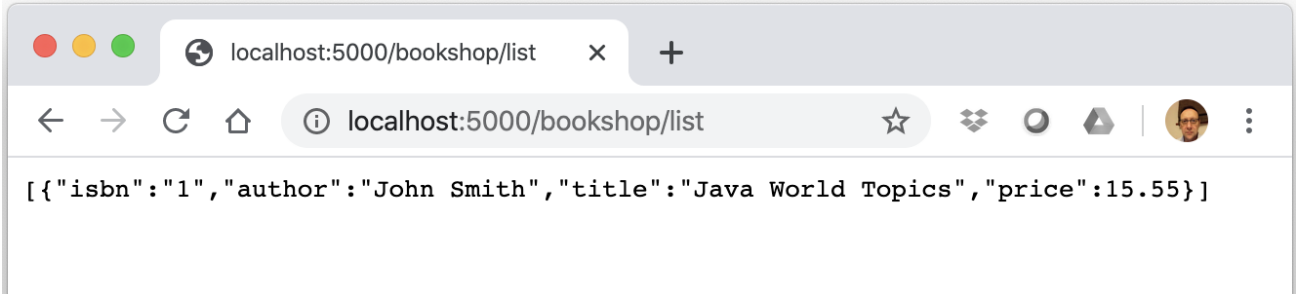
- Docker image available for RESTful bookshop service

```
docker run -p 5000:8080 johnehunt/spring-boot-docker-bookshop-
    webapp:0.0.1-SNAPSHOT
```



Accessing Service

- Access via URL http://localhost:5000/bookshop/list



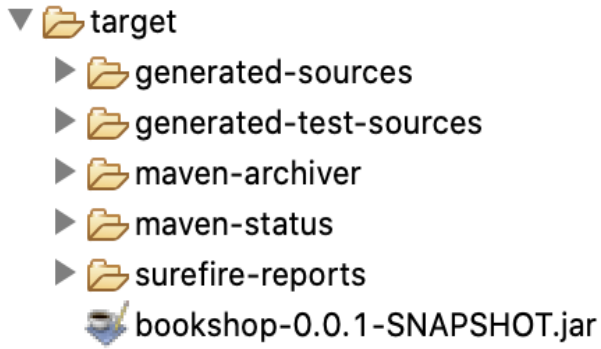
- Note Docker looked locally for image
  - if not found locally looked in central Docker Image hub / repo

Dockerizing Spring Boot App

- Create a Spring Boot web application
  - such as the Spring Boot Bookshop RESTful web services
  - available in a Git repo

```
git clone https://github.com/johnehunt/spring-boot-docker-
    bookshop-webapp.git
```

- Build the JAR file
  - will be used when we create the Docker image



Add the Dockerfile

- Provide a file called Dockerfile
  - at root of project
  - has a simple format
  - In effect this file contains a series of Docker commands that tell docker how to create the image

```
# Build on a base image with JDK
FROM openjdk:8-jdk-alpine

# Add Maintainer Info
LABEL maintainer="jjh@gmail.com"

# Add a volume pointing to /tmp
VOLUME /tmp

# Make port 8080 available
EXPOSE 8080

# The application's jar file
ARG JAR_FILE=target/bookshop-0.0.1-SNAPSHOT.jar
ADD ${JAR_FILE} bookshop.jar

# Run the jar file
ENTRYPOINT ["java","-
    Djava.security.egd=file:/dev/./urandom","-
    jar","/bookshop.jar"]
```

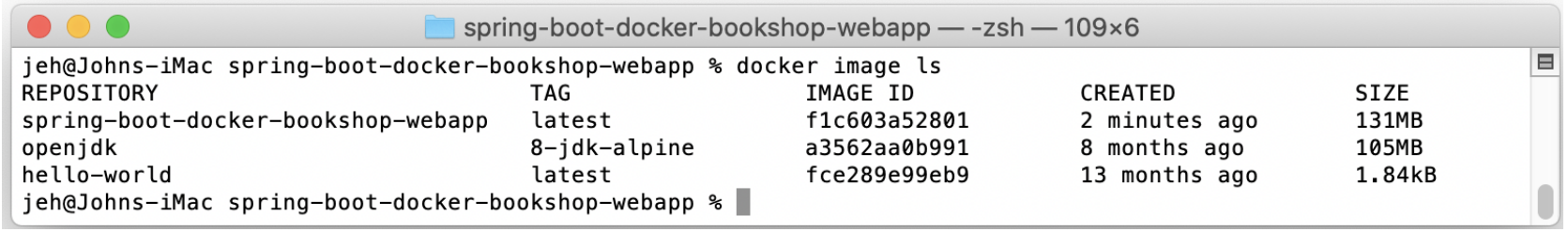
Build Docker Image

- Use the `docker build` command to build the image

```
docker build -t spring-boot-docker-bookshop-webapp .
```

- Uses the Dockerfile to build the image
- Can check whether image was created and installed locally

```
docker image ls
```



Run or Push the Docker Image

- Can now use docker to run application
  - as done earlier
- Can push Docker Image to Docker Hub
  - need to first login to the docker hub

```
docker login
```

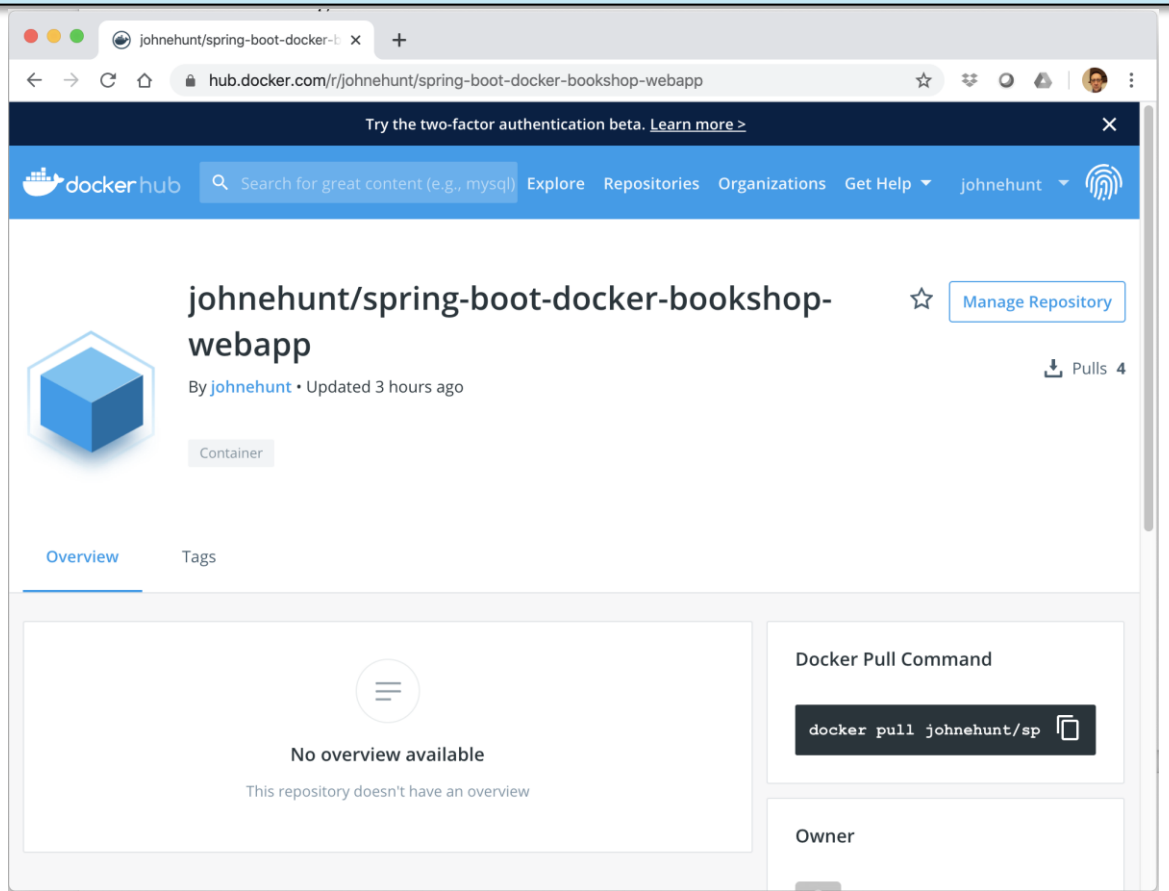
  - then tag the Docker image you created

```
docker tag spring-boot-docker-bookshop-webapp
   johnehunt/spring-boot-docker-bookshop-webapp:0.0.1-
   SNAPSHOT
```

Run or Push the Docker Image

- Finally push the image to the repository

```
docker push johnehunt/spring-boot-docker-bookshop-
   webapp:0.0.1-SNAPSHOT
```

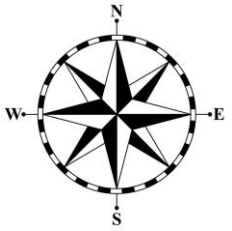


Databases

Toby Dussek

Informed Academy

Plan for Session



- What is a Database?
- Why use a Database?
- Relational Databases
- SQL and Databases
- Examples of Relational Databases
- NoSQL Databases
- Examples of NoSQL Databases
- Relational V NoSQL Databases

What is a Database?

?

What is a Database?

- A group / collection of data
  - organised in some way
  - to allow for future retrieval
- A Database Management System (DBMS)
  - software system that
  - manages one or more databases
  - stores and retrieves data in those databases
  - provides an interface allowing access to databases
  - may provide a GUI for databases
  - stores data typically in underlying file system
    - although some store data just in memory

Why use a Database?

- Shared Data
  - manages access to data by multiple users
  - typically manages concurrent access to data
- Performance
  - more efficient to search databases than plain text files
- Centralised Control
  - for access
  - for organisation
  - for standards (less so for NoSQL)

Relational Databases

- Data stored in tables

attribute

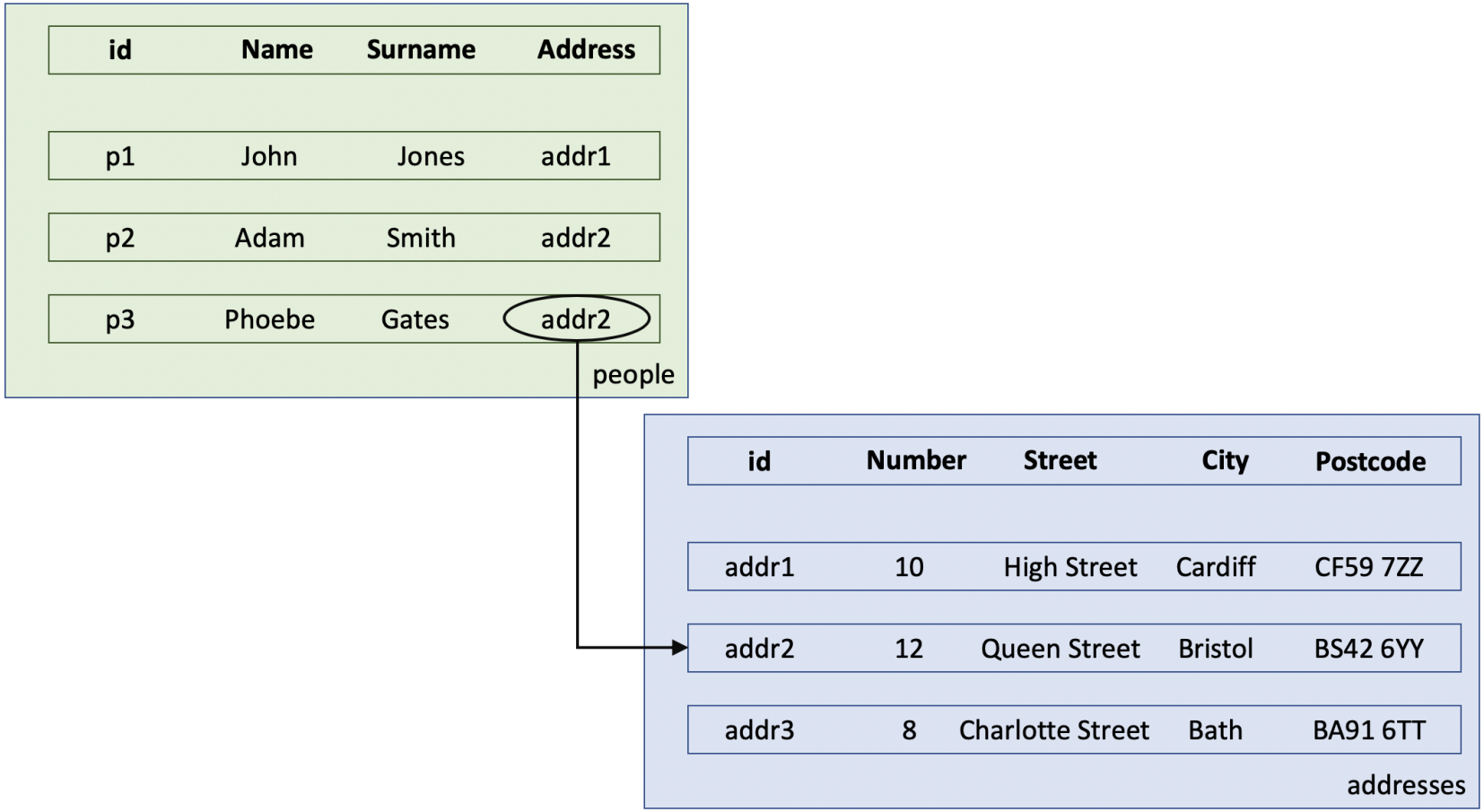
row

id	name	surname	subject	email
cs_18	Phoebe	Cooke	Animation	pc@my.com
cs_21	Gryff	Jones	Games	gj@my.com
cs_27	Adam	Fosh	Music	af@my.com
cs_29	Jasmine	Smith	Games	js@my.com

students

Relational Databases

- Data in one table can have relationships with another



Examples of Relational Databases

- MySQL
  - open source, free database

- Oracle Database
  - enterprise level commercial database

- Microsoft SQL Server
  - default in Microsoft world

- IBM DB2

  - another enterprise quality commercial database

Relational Databases

- Structure defined in a schema
  - using Data Definition Language (DDL) or GUI

Query 1 | new_table - Table

Name: students Schema: students

Column	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	G	Default / Expression
id	INT	☑	☑	☐	☐	☐	☐	☐	☐	
name	VARCHAR(45)	☐	☑	☐	☐	☐	☐	☐	☐	
surname	VARCHAR(45)	☐	☑	☐	☐	☐	☐	☐	☐	
subject	VARCHAR(45)	☐	☑	☐	☐	☐	☐	☐	☐	
email	VARCHAR(45)	☐	☑	☑	☐	☐	☐	☐	☐	

Column details 'email'

Column Name: email Datatype: VARCHAR(45)

Collation: Table Default Default

Comments: Storage: VIRTUAL STORED

Primary Key Not NU

Binary Unsigne

```
1  CREATE TABLE `students`.`students` (
2    `id` INT NOT NULL,
3    `name` VARCHAR(45) NOT NULL,
4    `surname` VARCHAR(45) NOT NULL,
5    `subject` VARCHAR(45) NOT NULL,
6    `email` VARCHAR(45) NOT NULL,
7    PRIMARY KEY (`id`),
8    UNIQUE INDEX `email_UNIQUE` (`email` ASC));
9
```

Structured Query Language

- SQL
  - standard language for relational databases
  - used to select data to retrieve
- SQL data definition language (DDL)
  - create, delete and update table schemas
- SQL data modification language (DML)
  - inserting, deleting, updating and querying table data
- SQL data control language (DCL)
  - grant/deny/revoke permissions

Structured Query Language

- Given a table such as

id	name	surname	subject	email
cs_18	Phoebe	Cooke	Animation	pc@my.com
cs_21	Gryff	Jones	Games	gj@my.com
cs_27	Adam	Fosh	Music	af@my.com
cs_29	Jasmine	Smith	Games	js@my.com
cs_31	Tom	Jones	Music	tj@my.com

student_table

```
SELECT * FROM students;
```

id	name	surname	subject	email
1	Phoebe	Cooke	Animation	pc@my.com
2	Gryff	Jones	Games	grj@my.com
3	Adam	Fosh	Music	af@my.com
4	Jasmine	Smith	Games	js@my.com
5	Tom	Jones	Music	tj@my.com
6	James	Andrews	Games	ja@my.com

Select

- Used to retrieve (select) data from the database
- Syntax

select	columns you wish to view
from	identifies the source tables to be examined
where	specifies a set of selection conditions
group by	group together rows for aggregation
having	restrict returned (not computed) rows
order by	sorts the result set

where Clause

- where limits the rows returned in a result set
  - only rows matching the where condition and returned

id	name	surname	subject	email
cs_18	Phoebe	Cooke	Animation	pc@my.com
cs_21	Gryff	Jones	Games	gj@my.com
cs_27	Adam	Fosh	Music	af@my.com
cs_29	Jasmine	Smith	Games	js@my.com
cs_31	Tom	Jones	Music	tj@my.com

student_table

```
SELECT * FROM students WHERE surname='Jones';
```

id	name	surname	subject	email
2	Gryff	Jones	Games	gj@my.com
5	Tom	Jones	Music	tj@my.com

where Clause

- Equality operators

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
=	equals
<> !=	not equal to

- Boolean operators

and	true if both operands are true
or	true if one or both operands are true
not	true if the operand is false

where Clause

- Using WHERE with equality and logical operators

```
SELECT * FROM students WHERE surname = 'Jones' and subject ='Music';
```

id	name	surname	subject	email
5	Tom	Jones	Music	tj@my.com
NULL	NULL	NULL	NULL	NULL

```
SELECT * FROM students WHERE surname = 'Jones' and id > 3;
```

id	name	surname	subject	email
5	Tom	Jones	Music	tj@my.com
NULL	NULL	NULL	NULL	NULL

order by Clause

- Sorts the results
  - into an ascending (default) or descending order

```
SELECT * FROM students ORDER BY name;
```

id	name	surname	subject	email
3	Adam	Fosh	Music	af@my.com
2	Gryff	Jones	Games	grj@my.com
6	James	Andrews	Games	ja@my.com
4	Jasmine	Smith	Games	js@my.com
1	Phoebe	Cooke	Animation	pc@my.com
5	Tom	Jones	Music	tj@my.com

```
SELECT * FROM students ORDER BY name desc;
```

id	name	surname	subject	email
5	Tom	Jones	Music	tj@my.com
1	Phoebe	Cooke	Animation	pc@my.com
4	Jasmine	Smith	Games	js@my.com
6	James	Andrews	Games	ja@my.com
2	Gryff	Jones	Games	grj@my.com
3	Adam	Fosh	Music	af@my.com

order by and where Clause

- Sorts the results
  - into an ascending (default) or descending order

```
SELECT * FROM students WHERE subject = 'Games' ORDER BY name;
```

id	name	surname	subject	email
2	Gryff	Jones	Games	grj@my.com
6	James	Andrews	Games	ja@my.com
4	Jasmine	Smith	Games	js@my.com

```
SELECT * FROM students WHERE subject = 'Games' ORDER BY name desc;
```

id	name	surname	subject	email
4	Jasmine	Smith	Games	js@my.com
6	James	Andrews	Games	ja@my.com
2	Gryff	Jones	Games	grj@my.com

Limit the selection number

- Partial result sets
  - return only the first n rows

```
SELECT * FROM students WHERE subject = 'Games' LIMIT 2;
```

id	name	surname	subject	email
2	Gryff	Jones	Games	grj@my.com
4	Jasmine	Smith	Games	js@my.com

Insert Data

- Use INSERT command to insert values into a table

INSERT INTO students (`id`, `name`, `surname`, `subject`, `email`)
VALUES (<{id: }>, <{name: }>, <{surname: }>, <{subject: }>, <{email: }>);

id	name	surname	subject	email
1	Phoebe	Cooke	Animation	pc@my.com
2	Gryff	Jones	Games	grj@my.com
3	Adam	Fosh	Music	af@my.com
4	Jasmine	Smith	Games	js@my.com
5	Tom	Jones	Music	tj@my.com
6	James	Andrews	Games	ja@my.com

```
INSERT INTO students (`id`, `name`, `surname`, `subject`, `email`)
      VALUES (7, 'Bill', 'Jones', 'Games', 'bj@my.com');
```

id	name	surname	subject	email
1	Phoebe	Cooke	Animation	pc@my.com
2	Gryff	Jones	Games	grj@my.com
3	Adam	Fosh	Music	af@my.com
4	Jasmine	Smith	Games	js@my.com
5	Tom	Jones	Music	tj@my.com
6	James	Andrews	Games	ja@my.com
7	Bill	Jones	Games	bj@my.com

Update Data

- Use UPDATE statement – with WHERE to indicate row or rows to be updated

```
UPDATE students
SET `id` = <{id: }>, `name` = <{name: }>, `surname` = <{surname: }>,
      `subject` = <{subject: }>, `email` = <{email: }>
WHERE `id` = <{expr}>;
```

```
UPDATE students
SET `name` = 'William'
WHERE `id` = 7;
```

id	name	surname	subject	email
1	Phoebe	Cooke	Animation	pc@my.com
2	Gryff	Jones	Games	grj@my.com
3	Adam	Fosh	Music	af@my.com
4	Jasmine	Smith	Games	js@my.com
5	Tom	Jones	Music	tj@my.com
6	James	Andrews	Games	ja@my.com
7	William	Jones	Games	bj@my.com

Delete Data

- Use DELETE statement to delete 1 or more rows

DELETE FROM students WHERE <{where_expression}>;

id	name	surname	subject	email
1	Phoebe	Cooke	Animation	pc@my.com
2	Gryff	Jones	Games	grj@my.com
3	Adam	Fosh	Music	af@my.com
4	Jasmine	Smith	Games	js@my.com
5	Tom	Jones	Music	tj@my.com
6	James	Andrews	Games	ja@my.com
7	William	Jones	Games	bj@my.com

```
DELETE FROM students WHERE `id` = 7;
```

id	name	surname	subject	email
1	Phoebe	Cooke	Animation	pc@my.com
2	Gryff	Jones	Games	grj@my.com
3	Adam	Fosh	Music	af@my.com
4	Jasmine	Smith	Games	js@my.com
5	Tom	Jones	Music	tj@my.com
6	James	Andrews	Games	ja@my.com

ACID

- Atomicity – all or nothing transactions
- Consistency – Guarantees Committed Transaction State
- Isolation – Transactions are Independent
- Durability – Committed Data is Never Lost

NoSQL Databases

- Really non-relational databases
  - Terminology originally referred to non-relational data stores
  - which did not provide an SQL interface
- Many different approaches now adopted
  - networks, lists, documents
  - a given NoSQL product may support multiple data models
- Typically provided client language bindings for their content
  - May provide an SQL interface

Examples of NoSQL Databases

- Document-oriented 

  - data encoded as discrete, self-contained documents
- Key-Value 

  - typically described as "schema-less"
- Graph 

  - data represented using nodes, edges and properties
  - typically direct pointers between adjacent elements

Relational V NoSQL Databases

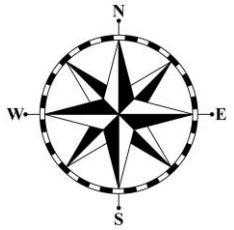
- Scalability
  - RDBs vertically scalable using CPU, RAM, SSD
  - NoSQL horizontally scalable by adding more servers
- Structure
  - RDBs better for multi-row transactional behaviour
    - such as accounting systems or online purchases
  - NoSQL where the structure is fluid or cannot be predicted
    - e.g. articles, social media posts, unstructured data
- ACID compliance
  - RDBs are ACID compliant / NoSQL tend not to be
  - Atomicity, Consistency, Isolation, Durability

Relational Database Design

Toby Dussek

Informed Academy

Plan for Session



- Data Relationships
- Database Design
- Database Normalisation
- First Normal From
- Second Normal From
- Third Normal From
- Other Database Features

Data Relationships

- **one : one**
  - 1 row in 1 table references 1 and only 1 row in another table
  - a person to an order for a unique piece of jewellery
- **one: many**
  - 1 row in 1 table references many rows in another table
  - all people living at the same address etc.
- **many : many**
  - many rows in 1 table reference many rows in a second table
  - many students may take a particular class and a student may take many classes

Database Design

- Identify Entities
  - key concepts about which data may be held
  - names, properties etc.
- Determine Tables based on Entities
- Identify Attributes based on entity properties
- Identify keys
  - primary keys, natural keys, artificial keys
- Identify Relationships
  - and how to model them

Database Normalisation

- Aims to refine a database design
  - revises how data is split across tables
  - to avoid data redundancy and maintain data integrity
- Several different Normal Forms
  - based on a set of standard (common sense) rules
  - 1NF (pronounced 1st Normal Form), 2NF to 6NF
  - typically designers focus on 1st to 3rd NF

First Normal Form

- Each attribute contains only a single value
- For example the following table is not in 1NF

id	name	surname	subject	email	modules
cs_18	Phoebe	Cooke	Animation	pc@my.com	Python
cs_21	Gryff	Jones	Games	gj@my.com	Databases OOP

student_table

First Normal Form

- We have now applied the 1NF

id	name	surname	subject	email
cs_18	Phoebe	Cooke	Animation	pc@my.com
cs_21	Gryff	Jones	Games	gj@my.com

student_table

id	student_id	module
1	cs_21	Databases
2	cs_21	OOP
3	cs_18	Python

module_table

First Normal Form

- Relationships between 1 table and another (1:Many)

id	name	surname	subject	email
cs_18	Phoebe	Cooke	Animation	pc@my.com
cs_21	Gryff	Jones	Games	gj@my.com

student_table

id	student_id	module
1	cs_21	Databases
2	cs_21	OOP
3	cs_18	Python

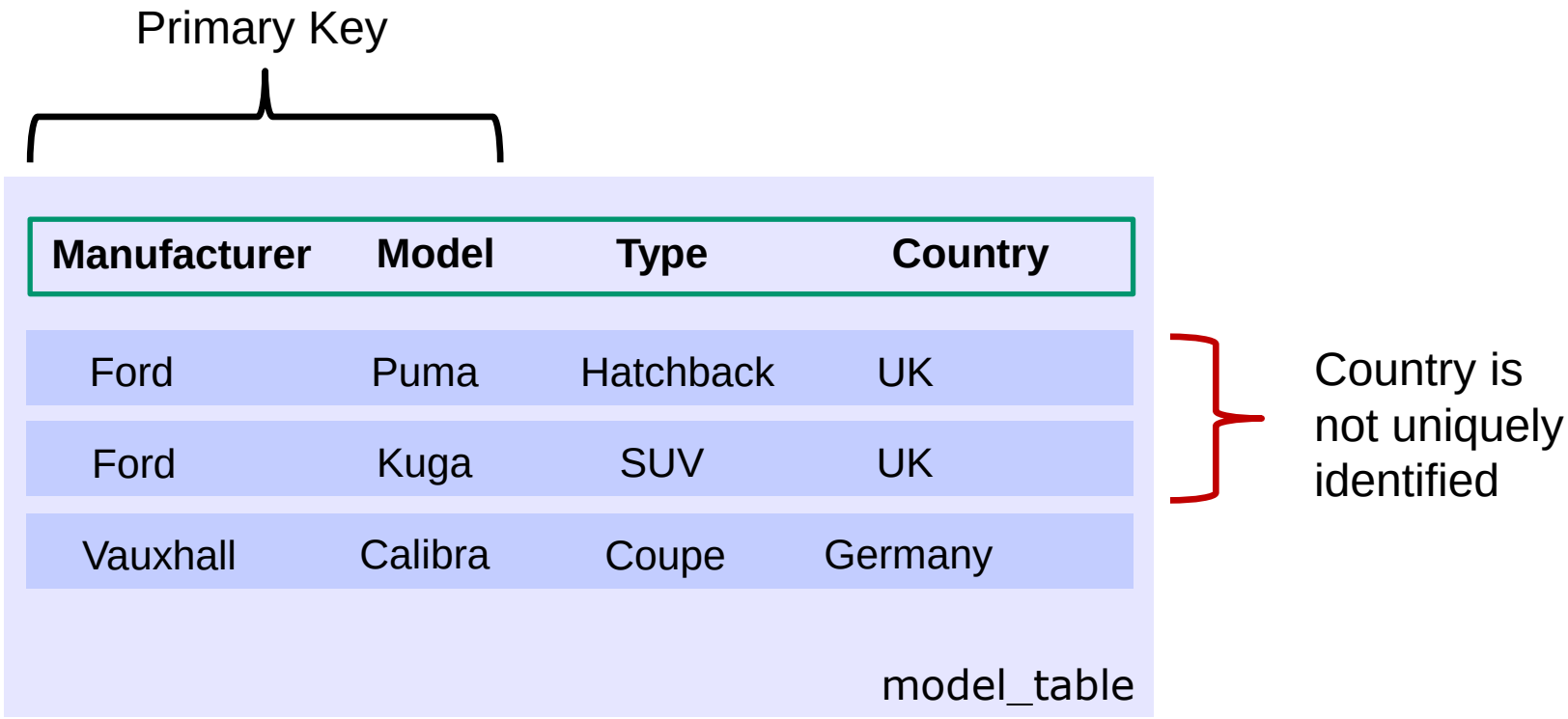
module_table

Second Normal Form

- Must be in 1NF
- Only relevant for composite primary keys
  - a primary key comprised of more than one attribute
- All non-primary attributes depend on the primary attributes
- That is, all the attributes that are not part of the primary key must depend on that primary key to be identified

Second Normal Form

- Not in 2NF



Second Normal Form

- Now in 2NF

Manufacturer	Country
Ford	UK
Vauxhall	Germany

manufacturer_table

Primary Key

Manufacturer	Model	Type
Ford	Puma	Hatchback
Ford	Kuga	SUV
Vauxhall	Calibra	Coupe

model_table

Third Normal Form

- Must be 2NF
- No non key attribute may depend on any other non key attribute
  - i.e. non key attributes only depend on keys

order_id	customer_id	product_id	quantity	product_name
245	162	23	15	Stapler
246	162	65	3	Hole Punch
247	34	21	5	Desk Lamp

order_table

product_name depends on product_id

Third Normal Form

- To represent info in 3NF; split the table into two

order_id	customer_id	product_id	quantity
245	162	23	15
246	162	65	3
247	34	21	5

order_table

product_id	product_name
23	Stapler
65	Hole Punch
21	Desk Lamp

product_table

Other Database Features

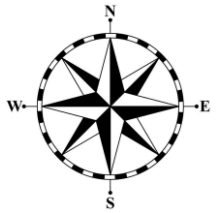
- Rules
  - define constraints on values / rows in a table
  - for example an voter cannot exist without a home address
- Index
  - a mechanism to provide a different way to organise rows in a database
  - can significantly improve performance
- View
  - an imaginary table created from other tables
  - can simplify queries and hide complex table structures
- Triggers behaviour executed by the DBMS

JDBC Overview

Toby Dussek

Informed Academy

Plan for Session



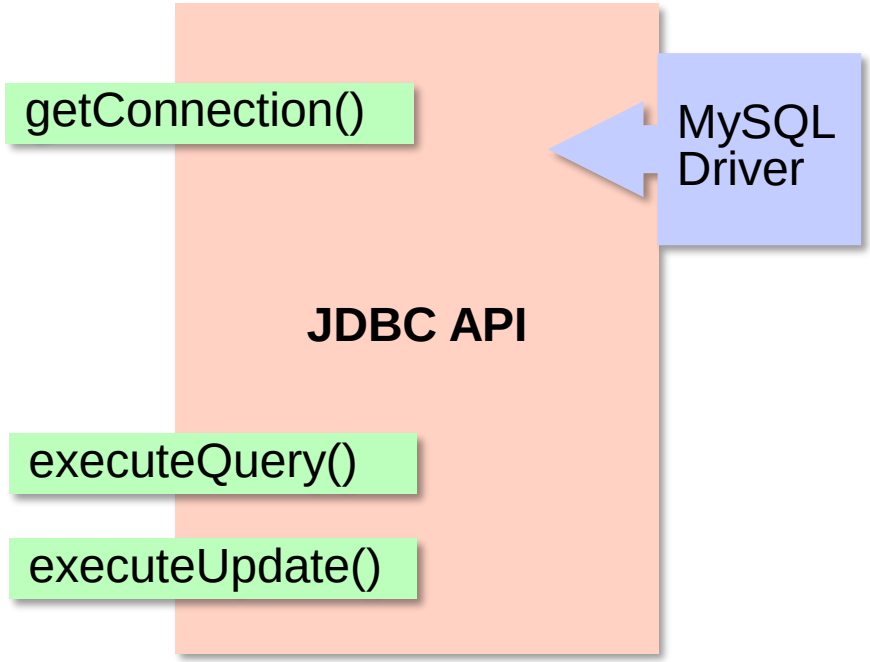
- JDBC Overview
- How JDBC Works
- Database Driver Management
- Four Types of Driver
- Obtaining a Driver
- Working with JDBC
  - opening a connection, creating a table, running a query
- Prepared Statements & Stored Procedures
- Transactions

JDBC Overview

- JDBC - Java Database Connectivity API
- Provides common front end for different databases
- Relies on database specific drivers for the back end
- Provides an ODBC bridge
- Can be used with Applications/Services/Servlets/JSPs
- Implemented by the java.sql and javax.sql packages
- Underpins all other Java Database access technologies
  - including Hibernate and JPA

How JDBC works

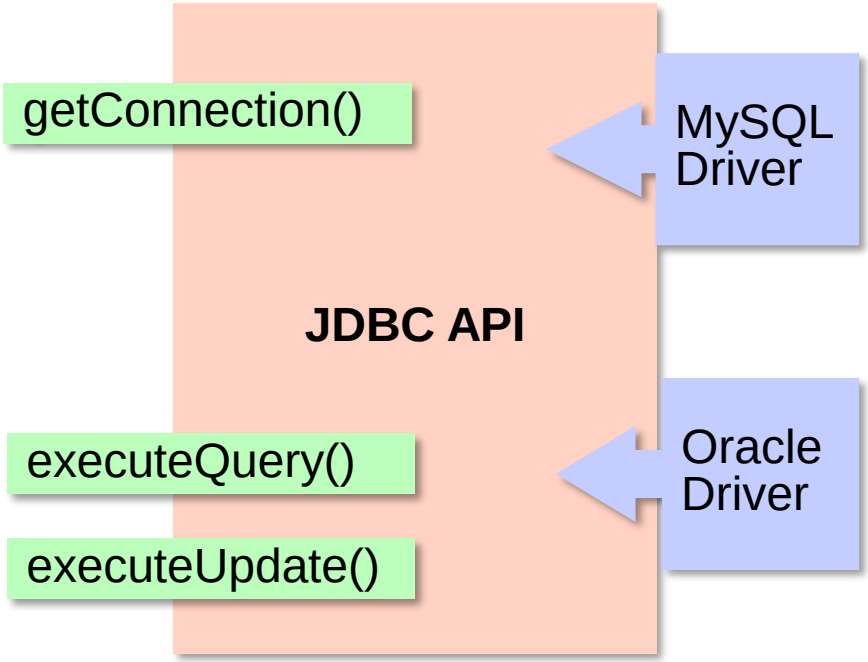
- Load a back-end database driver



- can be discovered automatically in recent versions
- Use common front end

How JDBC works

- Can load multiple database drivers



- Appropriate back end chosen for current database

Database Driver Management

- Connections handled by driver manager
- Historically must "register" a driver with the driver manager
  - latest JDBC can discover Driver Managers
- Simplest approach :
  - Programmatically by requesting the class of the driver

```
try {
    // The newInstance() call is a work around for some
    // broken Java implementations
    Class.forName("com.mysql.jdbc.Driver").newInstance();
} catch (Exception ex) {
    // handle the error
}
```

Four types of drivers

- Type 1: JDBC-ODBC Bridge
- Type 2: Java to Native API
- Type 3: Java to Middle ware
  - pure Java to middleware layer
- Type 4: Java to Native Database
  - pure Java directly to database
- Which is most appropriate depends on the application

Setting up the Project

- Will need the MySQL Driver on the classpath
  - can use Maven for this

```
<project … >
        <modelVersion>4.0.0</modelVersion>
        …

        <dependencies>
                <dependency>
                        <groupId>mysql</groupId>
                        <artifactId>mysql-connector-java</artifactId>
                        <version>8.0.20</version>
                </dependency>
                …
        </dependencies>
</project>
```

Opening a Connection

- Load the appropriate driver (e.g. ODBC)
- Make a connection through the DriverManager

```
String user = "user";
String pwd = "user123";
String url = "jdbc:mysql://localhost/uni-database";

Connection conn = DriverManager.getConnection(url,user,pwd);
```

- Format of URL:
  - `jdbc:<type of driver>:<driver specific info>`
- Another example (Oracle):
  - `"jdbc:oracle:thin:@localhost:1521:XE"`

Creating a Table

- Use a Statement object obtained from the Connection
  - Statement statement = con.createStatement();
- Use the executeUpdate() method
  - used for changes to database

```
String url = "jdbc:mysql://localhost/uni-database";
Connection conn = DriverManager.getConnection(url,"user", "user123");
Statement st = conn.createStatement();

st.executeUpdate("CREATE TABLE addresses (name char(15), address char(3))");
st.executeUpdate("INSERT INTO addresses (name, address) " +
                                "VALUES('John', 'C46')");
st.close();
conn.close();
```

  - make sure you close the statement and the connection
  - and handle any exceptions

Querying a Database

- Again use a Statement object
  - Use executeQuery() method on statement

```
try {
    String url = "jdbc:mysql://localhost/uni-database";
    Connection conn = DriverManager.getConnection(url,"user", "user123");
    Statement st = conn.createStatement();
    ResultSet rs = st.executeQuery("SELECT * FROM students");
    int cols = rs.getMetaData().getColumnCount();
    while (rs.next()) {
        for (int i = 1; i <= cols; i++) {
                System.out.print(rs.getString(i) + ", ");
        }
        System.out.println();
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

Querying a Database

- Two categories of get method
  - one positional
    - e.g. rs.getString(1);
  - one named
    - e.g. rs.getString("author");
- Different get methods for different types
  - getInt() / getString() / getBoolean() etc.

Prepared Statements

- Obtained from a connection object
- Inherits from Statement aimed at multiple executions
- Created with SQL which is sent to DBMS immediately to be compiled
- Can be used with parameters in SQL, indicated via ‘?’
  - INSERT INTO addresses (name, address) VALUES(?, ?)
- Use the setXXX(pos, value) methods
  - st.setString(1, "Phoebe");
  - st.setString(2, "B50");

Prepared Statement example

- Performs earlier example:

```
String sql = "INSERT INTO addresses (name, address) VALUES(?, ?)";
PreparedStatement st = con.prepareStatement(sql);
st.setString(1, "Adam");
st.setString(2, "A8");
st.executeUpdate();
st.setString(1, "Phoebe");
st.setString(2, "C7");
st.executeUpdate();
```

- Note zero parameter method executeUpdate method

Transactions

- Transaction treated as a unit
  - Either all execute or none execute
- Need to disable auto commit
  - con.setAutoCommit(false);
- Performed using the commit method
  - con.commit();
  - con.setAutoCommit(true);
- To rollback a set of operations
  - con.rollback();

Sample Transaction

```
con.setAutoCommit(false);
String sql = "INSERT INTO addresses (name, address)
   VALUES(?, ?)";
PreparedStatement st = con.prepareStatement(sql);
st.setString(1, "Adam");
st.setString(2, "A8");
st.executeUpdate();
st.setString(1, "Denise");
st.setString(2, "B56");
st.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

Stored Procedures

- Stored in database using executeUpdate
- Executed using a CallableStatement object

```
String storedProc = "…";
Statement st = con.createStatement();
st.executeUpdate(storedProc);
CallableStatement cst = con.prepareCall("{call …}");
ResultSet rs = cst.executeQuery();
```

JDBC is Not perfect

- Lowest common denominator approach
- Limited standardisation in Databases/ SQL outside its control
  - May result in database specific code
- Cursor used with results set can only move forward
- Drivers may have inherent limitations
- SQLException may be too gross a granularity
- For flexibility may need to use MetaData

Limitations of DriverManager

- Have to manually create a connection
  - connect to a database using an URL-like string

```
Connection con = DriverManager.getConnection("jdbc:odbc:Test");
```

- URL might specify a driver protocol, machine name, port number etc.
  - May be difficult to maintain
- What about connection pooling – need separate libraries!

Object Relational Mapping Introduction

Toby Dussek

Informed Academy

Persistence

- Persistence relates to

  The ability of an object to remain in existence past the lifetime of the program that creates it.

- This has been achieved in a variety of ways:
  - Flat files
  - Serialization
  - XML format (custom or Open Source)
  - JSON formats
  - Object Databases
  - Relational Databases

Paradigm Mismatch

- Problems caused when objects need to be stored in relational tables.
  - Java types v. SQL Datatypes
  - Granularity
  - Inheritance / Subtypes & Polymorphism
  - Identity
  - Associations
  - Graph Navigation
  - Cost

Problem of Granularity

- Objects can have various kinds of granularity (structure ??) to its data.
- Poor support for user defined types (UDT) in SQL
- No implementation is truly portable
  - SQL is not standard
  - different class structures may require different mapping
- May force developers to use less flexible representations on the object model.

Problem of Subtypes

- Java classes implement inheritance
  - (super & sub classes)
- Each sub or super class will define different data & completely different functionality
- Variables can hold objects of specified type or subtype (polymorphism)
- Relational model provides no support for inheritance or polymorphism

Problem of Identity

- Checking if objects are identical
- Java defines two notations:
  - Object identity [==]
  - Equality of value [.equals()]
- In SQL it is just checking if the two primary keys are the same.
- Two or more objects can represent the same row of data.
- Problem with SQL surrogate keys.

Problem of Association

- Associations represent relation between entities.
- Object references represents associations.
- In SQL association is by foreign keys
- Object references are directional, foreign keys are not.
- It isn’t possible to determine multiplicity of a association just by looking at a class.
- SQL multiplicity is always one-to-one or one-to-many, and can be determined by looking at the foreign key.

Cost Of The Mapping

- Requires significant time & effort
- Up to 30% of code is devoted to handling tedious SQL/JDBC
- Significant effort and time spent maintaining this code
- Logic of mapping spread across code base
  - Simple change in one model leads to extensive maintenance

Object/Relational Mapping (ORM)

- ORM is the automated & transparent persistence of Java objects to tables in a RDBMS
- Metadata is used to describe the mapping between the objects and the database
  - may be annotation based
  - or via external config file
- The ORM tool transforms data from one representation to another.

ORM Parts

- An ORM tool has the following aspects
  - API for CRUD operations on objects
    - **Create, read, update and delete** (**CRUD**) are the four basic functions of any storage system
  - API for queries for classes & properties
  - Facility for mapping metadata
  - Technique for interaction with transactional objects

Kinds of ORM

- There are four kinds of ORM solutions
  1. Pure relational
     - Whole app designed around the relational model
     - Lacks portability
  2. Light object mapping
     - Hand coded SQL/JDBC
     - Still very popular
  3. Medium object mapping
     - SQL generated at build time by tools or at run time by frameworks
  4. Full object mapping
     - Sophisticated object modeling: composition, inheritance, polymorphism & persistence.

What is Hibernate?

- Leading Open Source ORM Tools
  - See http://www.hibernate.org
  - current version of Hibernate is Version 5.4
- Can be used with any type of Java system
  - Desktop app, Servlets, Restful services, GraphQL, Java EE
  - Huge set of resources / examples / tutorials etc.
- Lots of books
  - Christian Bauer, Gavin King: *Hibernate In Action*, Manning Publications Company

What is JPA?

JPA
Java Persistence API

- Java Persistence API
  - https://www.oracle.com/java/technologies/persistence-jsp.html
  - several versions 1.0, 2.0, 2.1 and 2.2
- Single persistence API for Java EE and Java SE
  - "modelled" on Hibernate
- Several different implementations (all open source):
  - OpenJPA
  - DataNucleus
  - Hibernate wrapper
  - EclipseLink

Java Persistence API (JPA)

Toby Dussek

Informed Academy

Using JPA (and Hibernate)

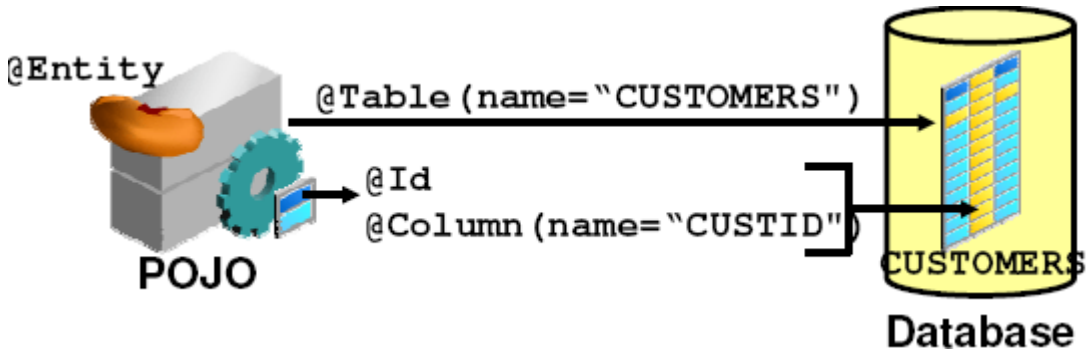
JPA
Java Persistence API
HIBERNATE

- JPA (Java Persistence API)
  - provides a specification / interface
  - for persisting, updating and deleting entities
  - for querying, finding objects in a Relational DB
  - with mapping meta data using Annotations
- Hibernate provides an implementation of JPA
  - widely used in Java / Spring world
  - both via JPA and its own Hibernate interface
- Major topics in their own right

What are JPA Entities?

JPA
Java Persistence API

- A JPA entity:
  - Is defined as a Plain Old Java Object (POJO)
  - marked with the Entity annotation
    - (no interfaces required)
  - Is mapped to a database using annotations



JPA aware POJO

JPA
Java Persistence API

- Add JPA Annotations
  - but note convention over configuration style
  - use @Transient to indicate a field that should NOT be persisted

```
package com.jjh.students;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="students")
public class Student {
        @Id
        // @GeneratedValue(strategy = GenerationType.AUTO)
        private int id;
        @Column(name="name")  // The default so unnecessary
        private String name;
        private String surname;
        private String subject;
        private String email;

        public Student() { } // Required by JPA

        public Student(int id, String name, String surname, String subject, String email) {
                this.id = id;
                this.name = name;
                this.surname = surname;
                this.subject = subject;
                this.email = email;
        }

        // ... Getters and Setters
}
```

Working with JPA

JPA
Java Persistence API

Simple steps:

- Set up dependencies
- Define a persistence context
- Access a configured Entity Manager
- Use Entity Manager API to
  - retrieve objects
  - add objects to database
  - update objects in database
  - remove objects from database

JPA Dependencies

- Need database driver, JPA specification and JPA Provider (e.g. Hibernate)

```
<dependencies>
   <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.20</version>
   </dependency>
   <dependency>
      <groupId>org.hibernate.javax.persistence</groupId>
      <artifactId>hibernate-jpa-2.1-api</artifactId>
      <version>1.0.2.Final</version>
   </dependency>
   <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-entitymanager</artifactId>
      <version>5.4.15.Final</version>
   </dependency>
</dependencies>
```

JPA
Java Persistence API

JPA Persistence Unit declaration

- META-INF/**persistence.xml** defines 1 or more persistence units

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
         http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
   version="2.0">

   <persistence-unit name="StudentJPA" transaction-type="RESOURCE_LOCAL">
      <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
      <properties>
         <property name="hibernate.connection.url" value="jdbc:mysql://localhost/uni-database" />
         <property name="hibernate.connection.driver_class" value="com.mysql.cj.jdbc.Driver" />
         <property name="hibernate.connection.username" value="user" />
         <property name="hibernate.connection.password" value="user123" />
         <property name="hibernate.archive.autodetection" value="class" />
         <property name="hibernate.show_sql" value="true" />
         <property name="hibernate.format_sql" value="true" />
         <property name="hbm2ddl.auto" value="update" />
      </properties>
   </persistence-unit>
</persistence>
```

What is the Entity Manager?

JPA
Java Persistence API

- Manages the life cycle of entity instances
  - creates and removes persistent entity instances,
  - finds entities by the entity's primary key, and
  - allows queries to be run on entities
- Associated with a persistence context
  - Defines classes and mappings used by Manager
  - Persistence units defined by persistence.xml configuration file
    - Located in META-INF or WEB-INF/classes

Using the Entity Manager

JPA
Java Persistence API

- Programmatically get an Entity manager

```
EntityManagerFactory emf =
                    Persistence.createEntityManagerFactory("StudentJPA");
EntityManager em = emf.createEntityManager();
```

- Or supplied by runtime environment
  - e.g. by Spring, WebLogic, WebSphere etc.
- An entity can be created by using:
  - The *new* operator (creates detached instance)
  - The EntityManager Query API (synchronized with the database)
  - An entity is inserted, updated, or deleted from a database through the EntityManager API.

EntityManager Methods

JPA
Java Persistence API

Commonly used methods include:

- **find(<class>, primary-key);**
  - Finds persisted object based on primary key
- **createQuery(JPQL, class)**
  - creates a TypedQuery object that returns result list
  - JPQL – JPA Query Language e.g. SELECT s FROM Student s
- **persist(Object obj);**
  - Saves / Updates the instance to the database.
  - In JPA-speak, it persists the instance using the JPA entity manager
- **remove(Object obj);**
  - Removes the persistence instance from the database.

Querying via JPA

JPA
Java Persistence API

- Can use standard SQL or JPQL
- JPQL replaces SQL and ORMs' own QL
  - Simplified syntax
    - Bulk update and delete operations
    - Projection list (SELECT clause)
    - Group by, Having
    - Subqueries (correlated and not)
    - Additional SQL functions
      - UPPER, LOWER, TRIM, CURRENT_DATE, ...
    - Dynamic queries
    - Polymorphic queries

Retrieving objects in JPA

JPA
Java Persistence API

- Results are instances of classes not result sets

```
EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("StudentJPA");
EntityManager em = emf.createEntityManager();

System.out.println("Find a student by ID");
Student student = em.find(Student.class, 1);
System.out.println(student);
```

```
Student(Phoebe, 1)
```

```
System.out.println("Final All students");
String jql = "SELECT e FROM Student e";
TypedQuery<Student> query = em.createQuery(jql, Student.class);
List<Student> results = query.getResultList();
for (Student s : results) {
    System.out.println(s);
}
```

```
Student(Phoebe, 1)
Student(Gryff, 2)
Student(Adam, 3)
Student(Jasmine, 4)
Student(Tom, 5)
Student(James, 6)
```

Retrieving objects in JPA

JPA
Java Persistence API

- Can use where clause to restrict objects returned

```
// Retrieve all students doing games
System.out.println("Final All students where subject is games");

String jql2 = "SELECT e FROM Student e WHERE e.subject = 'Games'";

TypedQuery<Student> query2 = em.createQuery(jql2, Student.class);

List<Student> results2 = query2.getResultList();
for (Student s : results2) {
        System.out.println(s);
}
```

```
Student(Gryff, 2)
Student(Jasmine, 4)
Student(James, 6)
```

Modifying Entities

JPA
Java Persistence API

- Can also add new entities via the EntityManager

```
em.getTransaction().begin(); // Note use of a transaction
Student s1 = new Student(7, "Bill", "John", "Games", "bj@my.com");
em.persist(s1);
em.getTransaction().commit();
```

- Update entities

```
Student s2 = em.find(Student.class, 7);
s2.setName("William");
em.getTransaction().begin();
em.persist(s2);
em.getTransaction().commit();
```

- As well as delete entities from the database

```
Student s3 = em.find(Student.class, 7);
em.getTransaction().begin();
em.remove(s3);
em.getTransaction().commit();
```

JPA Mapping Features

Toby Dussek

Informed Academy

Associations

- OO Designs have a number of associations
  - One-to-one, one-to-many, many-to-one, many-to-many, parent-child
- JPA provides support for these
  - Contained – components
  - Collections – bag, idbag, set, map, list etc.
  - One-to-one, One-to-many and many-to-one
- Will look at some of these
- But a major subject in its own right
  - Hibernate say – need to be an expert in Databases, OO and mappings

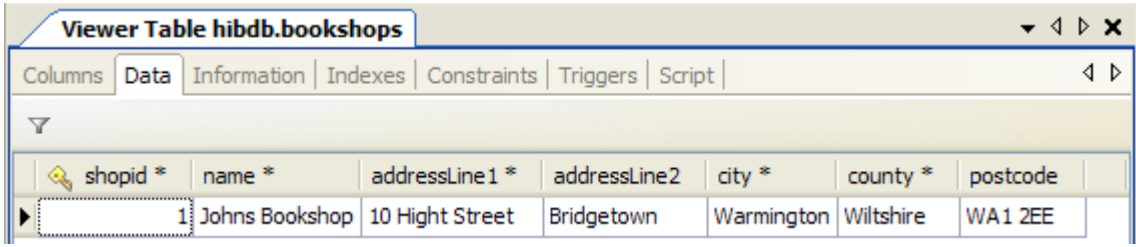
Mapping Annotations

- All in the javax.persistence package
- Field Annotations
  - Embedded – indicates containment
  - ElementCollection – used with collections such as sets
  - OneToMany - one-to-many multiplicity
  - ManyToOne – many-to-one multiplicity
  - ManyToMany - many-to-many multiplicity
  - OrderBy – specifies the ordering of elements in a collection
- Associated enum types
  - CascadeType – how to cascade operations
  - FetchType – supports Lazy and Eager fetch strategies

Mapping Contained Objects

- Bookshop has an Address object
- Address is a *component* of Bookshop
  - Address data wholly owned by Bookshop

```
@Entity
@Table(name="bookshops")
public class Bookshop {

    @Id
    @Column(name="shopid", nullable=false, columnDefinition="integer")
    private int id;
    @Column(nullable=false)
    private String name;
    @Embedded
    private Address address;
```



One to Many Relationship

- Holds an id, a name and a Set containing String values

```
public class Employee {
    private int id;
    private String name;
    private Set<String> languages = new HashSet<String>();
    public Employee() {}
    public Employee(int id, String name) {
        super();
        this.id = id;
        this.name = name;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public boolean addLanguage(String e) {
        return languages.add(e);
    }
    public boolean removeLanguage(String o) {
        return languages.remove(o);
    }
```

Database Tables

- Two tables, employee and languages
  - empid used as key

empid *	name *
1	John

empid *	language *
1	C#
1	C++
1	Java
1	JavaScript
1	jQuery
1	PHP
1	Scala

Element Collection Mapping

- Now use a @ElementCollection – as it is a contained relationship
  - Use collection table to provide the mapping info

```
@Entity(name="employee")
public class Employee {

    @Id
    @Column(name="empid")
    private int id;
    @Column(nullable=false)
    private String name;

    @ElementCollection(fetch=FetchType.LAZY)
    @CollectionTable(name="languages", joinColumns=@JoinColumn(name="empid"))
    @Column(name="language")
    private Set<String> languages = new HashSet<String>();
```

Map based mapping

- Can still use @ElementCollection but need some addition info re the keys

```
@Entity(name="companies")
public class Company {

    @Id
    @Column(name="coid")
    private int id;
    @Column(nullable=false)
    private String name;
    private String url;

    @ElementCollection(fetch=FetchType.LAZY)
    @MapKeyColumn(name="officeName")
    @CollectionTable(name="offices", joinColumns=@JoinColumn(name="coid"))
    @AttributeOverrides({
          @AttributeOverride(name="value.addressLine1", column=@Column(nullable=false)),
          @AttributeOverride(name="value.city", column=@Column(nullable=false)),
          @AttributeOverride(name="value.county", column=@Column(nullable=false))
       })
    private Map<String, Office> offices = new HashMap<String, Office>();
```

A company has many offices

Database Tables

- Again two tables but mapped objects have multiple properties
  - coid used as key between tables
  - officeName used as map key

coid *	name *	url
1	John CO.	www.john.co.uk

coid *	addressLine1 *	addressLine2	addressLine3	city *	county *	postcode	officeName *
1	Eagle Business Park	Ely		Cardiff	South Glamorgan	CF3 5RR	regional
1	10 High Street	Little Compton		Bath	BANES	BA1 4RT	main

One to Many List-based relationship

- The sequence in a List can be mapped in two different ways:
  - as ordered lists, where the order is not materialized in the database
  - as indexed lists, where the order is materialized in the database
- Can use @One-To-Many plus
  - Use an @OrderBy annotation on a property to order lists
  - Use @OrderColumn to specify an index

The Annotated classes

```java
@Entity(name="pupils")
public class Pupil {

    @Id
    private int id;
    private String name;
    @OneToMany(cascade=CascadeType.ALL)
    @OrderColumn(name="subject_index")
    private List<Subject> subjects = new ArrayList<Subject>();

    public Pupil() {}

    public Pupil(int id, String name) {
        this.id = id;
        this.name = name;
    }
```

```java
@Entity(name="subjects")
public class Subject {

    @Id
    private int id;
    private String title;
    private int credits;

    public Subject() {
    }
    public Subject(int id, String title, int credits) {
        super();
        this.id = id;
        this.title = title;
        this.credits = credits;
    }
```

Relational model

pupils table

id *	name
1	Adam

pupil_subjects table

pupils_id *	subjects_id *	subject_index *
1	1	0
1	2	1
1	3	2

subjects table

id *	credits *	title
1	20	English
2	20	Maths
3	10	DT Food

Annotation Based Many to One

```java
@Entity
@Table(name="customers")
public class Customer {

    @Id
    @Column(name="custid")
    private int id;
    @Column(nullable=false)
    private String name;
    @ManyToOne(fetch=FetchType.EAGER, cascade=CascadeType.ALL)
    @JoinColumn(name="billingAddressId", nullable=false)
    private BillingAddress billingAddress;

    public Customer() { }
```

Between Customers and BillingAddress

```java
@Entity(name="addresses")
public class BillingAddress {
    @Id
    @Column(name="billingAddressId")
    private int id;
    private String addressLine1;
    private String addressLine2;
    private String city;
    private String county;
    private String postcode;
    private String phone;

    public BillingAddress() {}
```

Relational Model

customers table

custid *	name *	billingAddressId *
1	John	0
2	Denise	0

addresses table

billingAddress...	addressLine1	addressLine2	city	county	phone	postcode
0	20 Walston Drive	Port George	Chippenham	Wiltshire	(01249) 56411112	SN10, 4YT

Many-to-Many Annotation

```java
@Entity
public class Event {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="event_ID")
    private Long id;

    private String title;

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name="event_date")
    private Date date;
    @ManyToMany
    private Set<Attendee> attendees = new HashSet<Attendee>();

    public Event() {}
```

Events has a many-to-many relationship with Attendees

```java
@Entity
@Table(name="attendees")
public class Attendee {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="attendee_id")
    private Long id;
    @Basic
    private int age;
    private String firstname;
    private String lastname;
    @ManyToMany
    private Set<Event> events = new HashSet<Event>();

    public Attendee() {}
```

Relational Model

event_ID *	event_date	title
1	17/04/2012 14:05:16	event1
2	17/04/2012 14:05:16	event2

attendees_attendee_id *	events_event_ID *
1	1
2	1
1	2

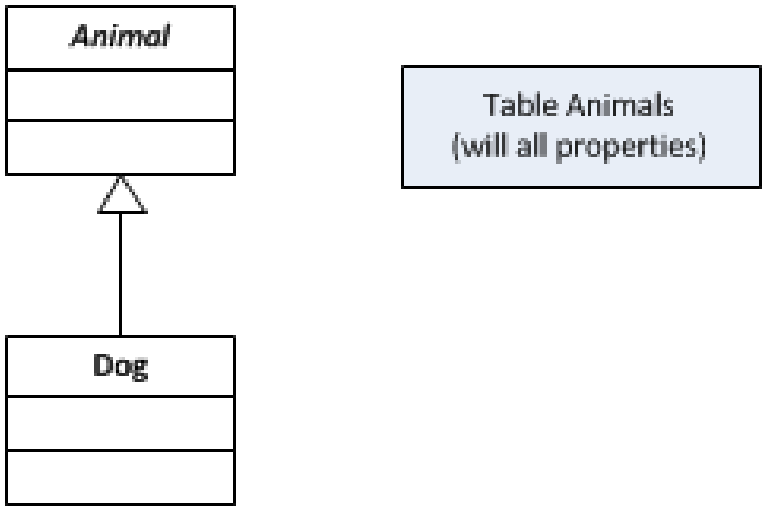
attendee_id *	age *	firstname	lastname
1	47	John	Hunt
2	45	Denise	Cooke

Inheritance

- Java has it but Relational DBs don't
- Various strategies
  - Table per concrete class
    - Simplest approach
  - Single Table per class hierarchy
  - Table per subclass (AKA JOINED)
    - Represents is-a as a has-a relationship
- Can mix within mapping meta data

Table Per Class hierarchy

- Includes columns for all properties of all classes in hierarchy
- Simple and fast



- Columns must be nullable
- Issue of extension

Annotations for Table Per Class Hierarchy

- Table and Inheritance strategy specified on root of class hierarchy (note the discriminator)

```
@Entity
@Table(name="animal")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="discriminator")
public class Animal {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    private String colour;

    public Animal() { }
```

Discriminator distinguishes actual class

```
@Entity
public class Dog extends Animal {

    private String name;

    public Dog() { }
```

discriminator *	id *	colour	name
Dog	1	Brown	Fido

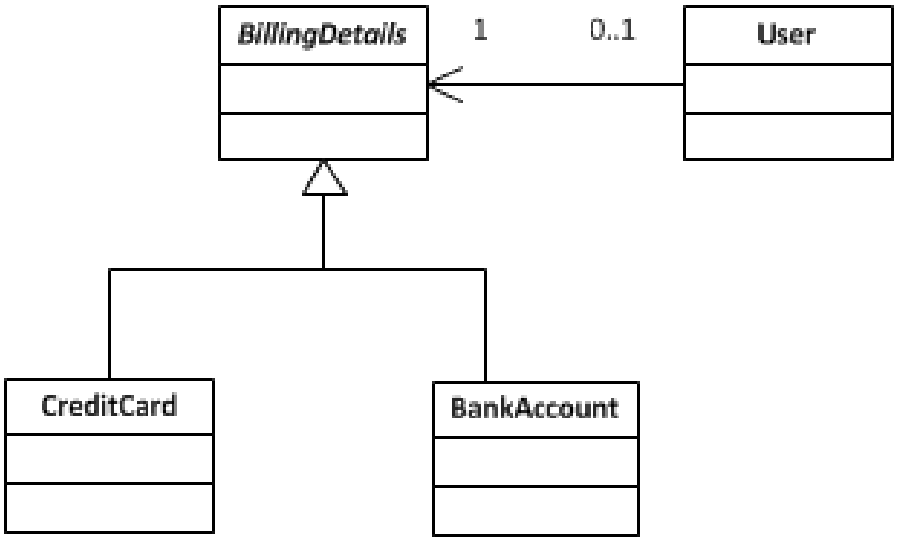
Loading

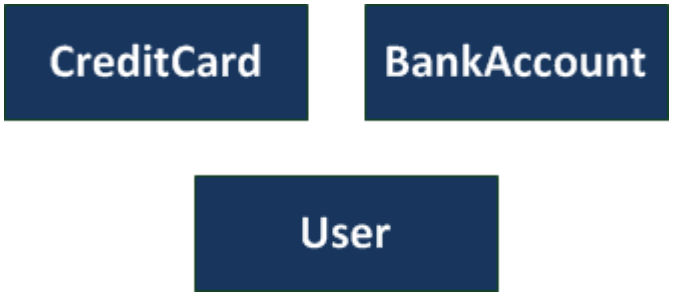
- Just reference root type – appropriate class instantiated

```
public class MainLoad {

    public static void main(String[] args) {
        Session session = HibernateSessionFactory.getSessionFactory();
        session.beginTransaction();

        System.out.println("Loading Company");
        System.out.println("--------------------------------");
        Animal animal = (Animal) session.get(Animal.class, 1);

        System.out.println(animal);
        if (animal instanceof Dog) {
            System.out.println("Its a Dog!");
        }

        System.out.println("--------------------------------");
        session.close();
    }

}
```

Table per Concrete class

- No special Hibernate constructs required
  - (although can use <union-subclass> mapping)





- But issues with polymorphic relationships
  - So may define a polymorphic association using <any>

Annotating for Polymorphism

- Users hold a reference to BillingDetails
- But BillingDetails is an abstract class at root of hierarchy

```
@Entity
@Table(name="users")
public class User {

    @Id
    @Column(name="userid")
    private int id;
    @Column(nullable=false)
    private String name;
    @Column(nullable=false)
    private String password;
    @Column(nullable=false)
    private String email;
    @ManyToOne(cascade=CascadeType.ALL)
    private BillingDetails billingDetails;
```

Annotation based Table per Concrete class

- Annotate root of hierarchy with @Inheritance and strategy

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class BillingDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private int id;
    @Column(nullable=false)
    private String number;
```

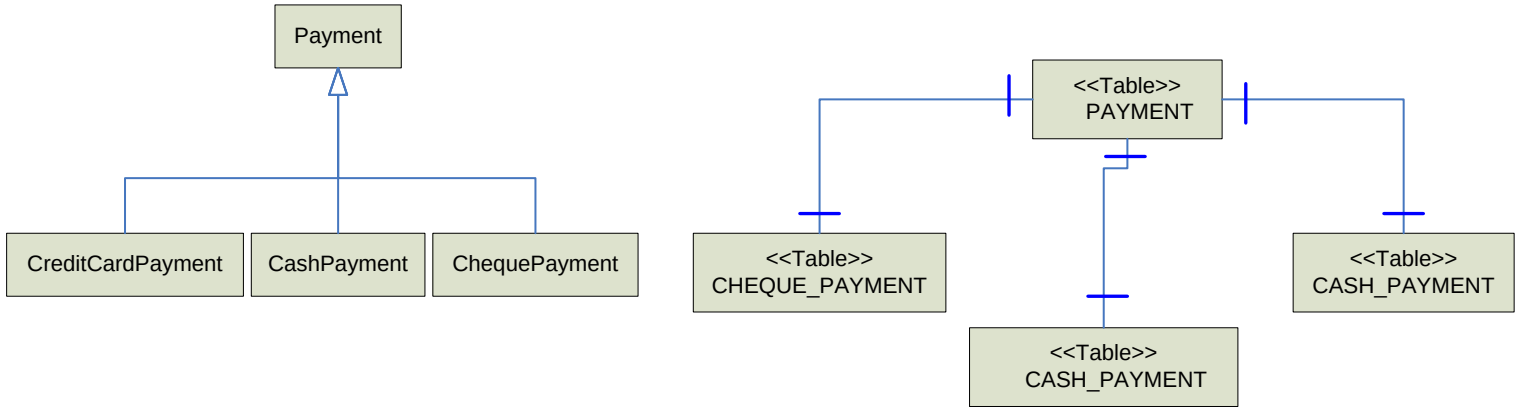
```
@Entity
@Table(name="bankAccount")
public class BankAccount extends BillingDetails {

    private String bankName;
```

```
@Entity
@Table(name="creditCard")
public class CreditCard extends BillingDetails {

    private int type;
    private String expMonth;
    private String expYear;
```

Subclasses specify their own tables

Table per Subclass

- Inheritance represented as relational foreign key
- All classes in hierarchy have a table
- Create an instance of a subclass – rows added to subclass table and to parent tables
- Performance may be an issue



Annotation mapping for Table per Subclass (or JOINED)

```
@Entity
@Table(name="student")
@Inheritance(strategy=InheritanceType.JOINED)
public class Student {
    @Id
    @Column(name="Student_ID")
    @GeneratedValue(strategy = GenerationType.TABLE)
    private Long studentId;
    private String firstname;
    private String lastname;
```

```
@Entity
@Table(name="undergraduates")
@PrimaryKeyJoinColumn(name="student_id",
                      referencedColumnName="student_id")
public class Undergraduate extends Student {

    private String subject;
```

PrimaryKeyJoinColumn annotations indicate how to join each subclass record to the corresponding record in its direct superclass

```
@Entity
@Table(name="postgraduates")
@PrimaryKeyJoinColumn(name="student_id",
                      referencedColumnName="student_id")
public class Postgraduate extends Student {

    private String research;
```

Relational model

Student table

Student_ID *	firstname	lastname
1	John	Hunt
2	Denise	Cooke

Undergraduates table

subject	student_id *
Software Engineering	1

Postgraduates table

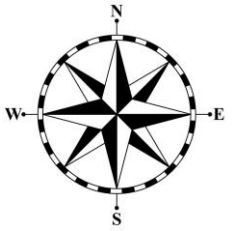
research	student_id *
Physics	2

Spring Data and JPA

Toby Dussek

Informed Academy

Plan for Session



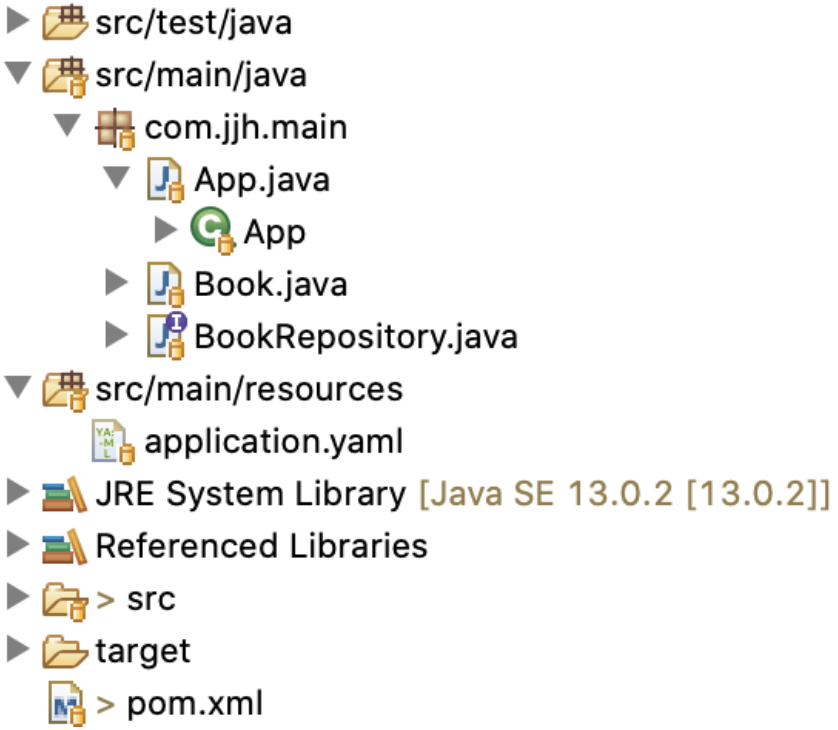
- Spring Data JPA Support
- JPA Setup
- Spring Repository
- Crud Repository Methods

Spring Data JPA Support

- Most JPA related code is boiler plate code
- Do the same thing to
  - create / insert
  - update
  - delete
  - find, findById, findAll
- Whether it's a Book, User, Trade, Product etc
- Could something handle this for us?

JPA Repository Based Project

- Spring Data massively simplifies working with JPA
  - use the spring-boot-starter-data-jpa dependency



JPA Repository

- Need to set up an *entity*
  - a POJO that will be persisted
  - using @Entity annotation &
  - probably @Id (others default)

```
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
@Table(name="books")
public class Book {
        @Id private String isbn;
        private String author;
        private String title;
        private double price;

        public Book() {}
        public Book(String isbn, String author,
                String title, double price) {
                this.isbn = isbn;
                this.author = author;
                this.title = title;
                this.price = price;
        }
```

JPA Properties

- Indicate what database, JPA implementation etc
- Can be properties or YAML file
  - e.g. application.yaml

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/bookshop
    password: user123
    username: user
    driver-class-name: "com.mysql.cj.jdbc.Driver"
  jpa:
    database-platform: org.hibernate.dialect.MySQL5InnoDBDialect
    hibernate:
      ddl-auto: update
```

Defining the Repository

- Spring provides repository for default behaviour
  - annotate *interface* with `@Repository` and extend `CrudRepository`
  - generic types provides details of type (class and key type)
  - Spring generates appropriate implementation for you
  - Uses JPA and a JPA implement (e.g. Hibernate)

```
package com.jjh.main;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface BookRepository extends CrudRepository<Book, String> {}
```

Using a Repo-sitory

```
package com.jjh.main;
import java.util.Optional;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication
public class App {
	public static void main(String[] args) {
		ConfigurableApplicationContext context =
				SpringApplication.run(App.class, args);
		BookRepository repository = context.getBean(BookRepository.class);

		Iterable<Book> books = repository.findAll();
		System.out.println("books: " + books);

		Book book = new Book("10", "Java and Spring", "John Smith", 12.55);
		repository.save(book);

		System.out.println("Check book with ISBN 10 is in database");
		Optional<Book> optionalBook = repository.findById("10");
		System.out.print(optionalBook);
		System.out.println("Delete Book just Added");
		repository.deleteById("10");
		System.out.println("Done");
	}
}
```

Running the JPA Repos Example

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.2.1.RELEASE)

2019-11-18 15:57:15.438  INFO 35116 --- [           main] com.jjh.main.App                         :
...
2019-11-18 15:57:15.960  INFO 35116 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate :
Bootstrapping Spring Data repositories in DEFAULT mode.
2019-11-18 15:57:15.997  INFO 35116 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate :
Finished Spring Data repository scanning in 30ms. Found 1 repository interfaces.
2019-11-18 15:57:16.587  INFO 35116 --- [           main] o.s.web.context.ContextLoader            : Root
WebApplicationContext: initialization completed in 1109 ms
2019-11-18 15:57:16.749  INFO 35116 --- [           main] o.hibernate.jpa.internal.util.LogHelper  :
HHH000204: Processing PersistenceUnitInfo [name: default]
2019-11-18 15:57:16.790  INFO 35116 --- [           main] org.hibernate.Version                    :
HHH000412: Hibernate Core {5.4.8.Final}
2019-11-18 15:57:16.872  INFO 35116 --- [           main] o.hibernate.annotations.common.Version   :
HCANN000001: Hibernate Commons Annotations {5.1.0.Final}
2019-11-18 15:57:17.236  INFO 35116 --- [           main] org.hibernate.dialect.Dialect            :
HHH000400: Using dialect: org.hibernate.dialect.MySQL5InnoDBDialect
2019-11-18 15:57:17.674  INFO 35116 --- [           main] o.h.e.t.j.p.i.JtaPlatformInitiator       :
HHH000490: Using JtaPlatform implementation:
[org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform]
books: [Book [id=5, author=Bob, title=Java and Spring, price=23.66], Book [id=6, author=Adam, title=Java
by Night, price=3.44]]
Check book with ISBN 10 is in database
Optional[Book [id=10, author=Java and Spring, title=John Smith, price=12.55]]Delete Book just Added
Done
```

CrudRepository Methods

long	**count**()Returns the number of entities available.
void	**delete**(**T** entity)Deletes a given entity.
void	**deleteAll**()Deletes all entities managed by the repository.
void	**deleteAll**(**Iterable**<? extends **T**> entities)Deletes the given entities.
void	**deleteById**(**ID** id)Deletes the entity with the given id.
boolean	**existsById**(**ID** id)Returns whether an entity with the given id exists.
Iterable<**T**>	**findAll**()Returns all instances of the type.
Iterable<**T**>	**findAllById**(**Iterable**<**ID**> ids)Returns all instances of the type T with the given IDs.
Optional<**T**>	**findById**(**ID** id) Retrieves an entity by its id.
S	**save**(S entity) Saves a given entity.
Iterable<S>	**saveAll**(**Iterable**<S> entities) Saves all given entities.

Defining the Repository

- Can create custom queries for repository
  - can use SQL (native) or JPQL

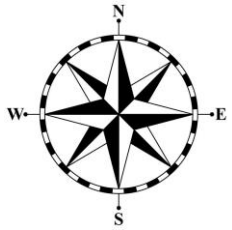
```
package com.jjh.main;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface BookRepository extends CrudRepository<Book, String> {

    @Query(value = "SELECT * FROM books WHERE price=0.0",
                nativeQuery = true)
    Collection<Book> findAllFreeBooks();

    @Query(value="SELECT * FROM books WHERE author = ?1 and title = ?2",
                nativeQuery = true)
    Book findBookByAuthorAndTitle(String author, String title);

}
```

Jenkins

Toby Dussek

Informed Academy

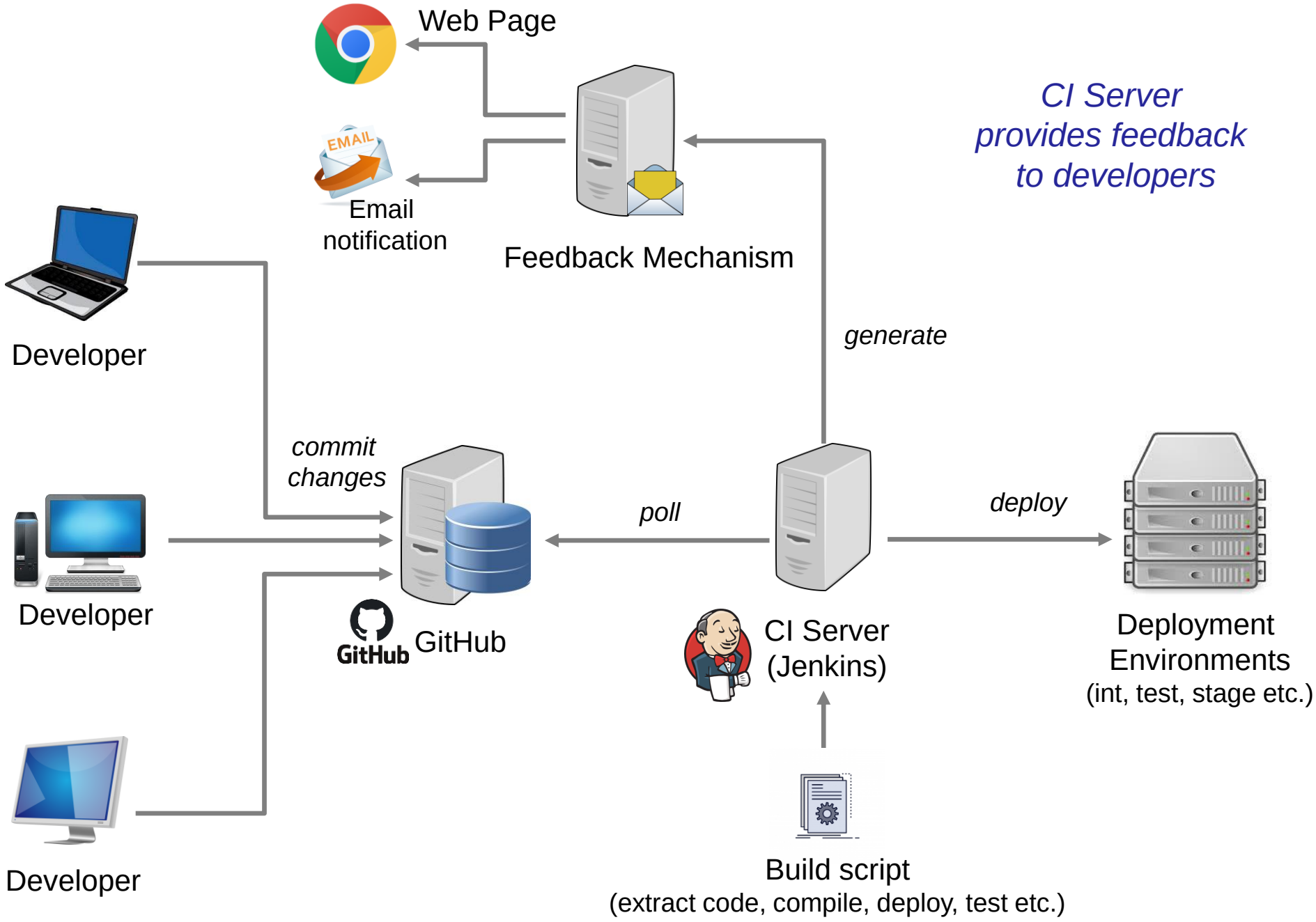
Plan for Session



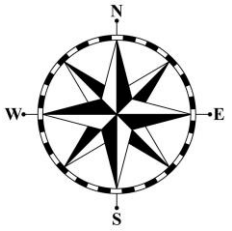
- What is Jenkins?
- Where Jenkins fits in
- Jenkins CI Environment
- Installing Jenkins
- Setting up Jenkins jobs

What is Jenkins?

- A Continuous Integration Server
- Very widely used open source automation server
- A pluggable system for building & deploying software
- Free to use
- Self contained
- Distributable build system
- A Java Web Application
- Extensively Documented
  - https://www.jenkins.io/


Web Page
EMAIL
Email notification
Feedback Mechanism
CI Server provides feedback to developers
Developer
generate
commit changes
poll
deploy
Developer
GitHub
GitHub
CI Server (Jenkins)
Deployment Environments
(int, test, stage etc.)
Developer
Build script
(extract code, compile, deploy, test etc.)

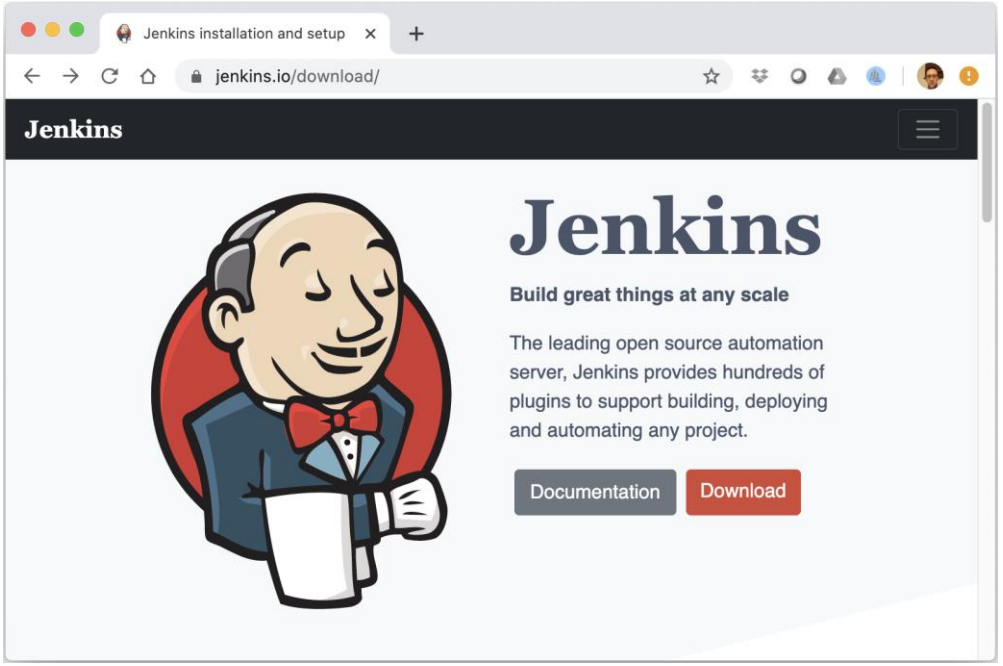
Setting up CI Environment



- Install Jenkins
- Install Required Plugins
- Set up a Build job
- Run build jobs
- View Results
- Jenkins Pipelines

Install Jenkins

- Download Jenkins from
  - https://www.jenkins.io/download/
  - select platform installer or Generic WAR file
  - requires Java to run as a Java Web App



Install Jenkins

- See documentation for setting up and running Jenkins
  - https://www.jenkins.io/doc/
- If running the WAR file
  - open a command window
  - change directory to where you have saved the jenkins.war file
  - run

```
java -jar jenkins.war --httpPort=8080
```

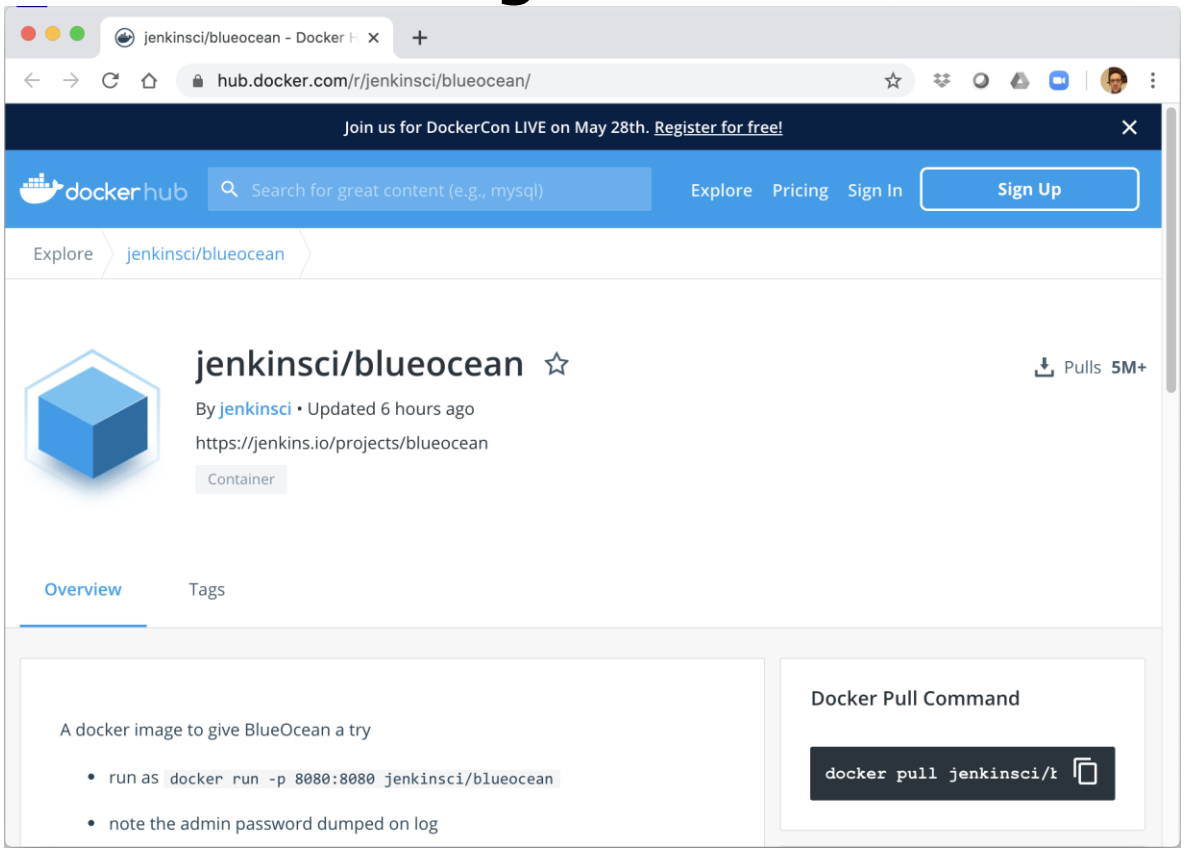
  - may need to enable future Java version

```
java -jar jenkins.war --httpPort=8080 --enable-future-java
```

  - navigate to http://localhost:8080 to complete installation

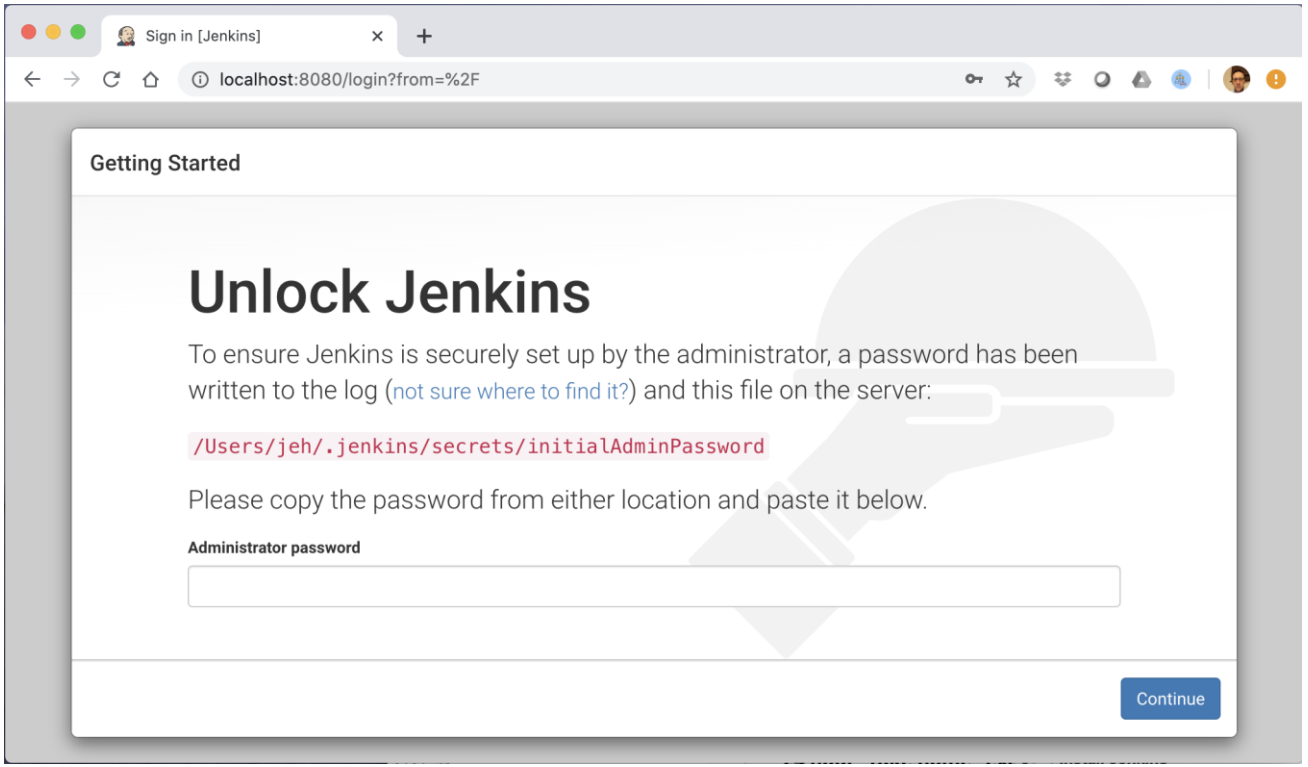
Install Jenkins

- Alternatively can use Docker image
- Several provided
  - see Docker Hub repository
  - use is the jenkinsci/blueocean image
  - contains current LTS version of Jenkins



Install Jenkins

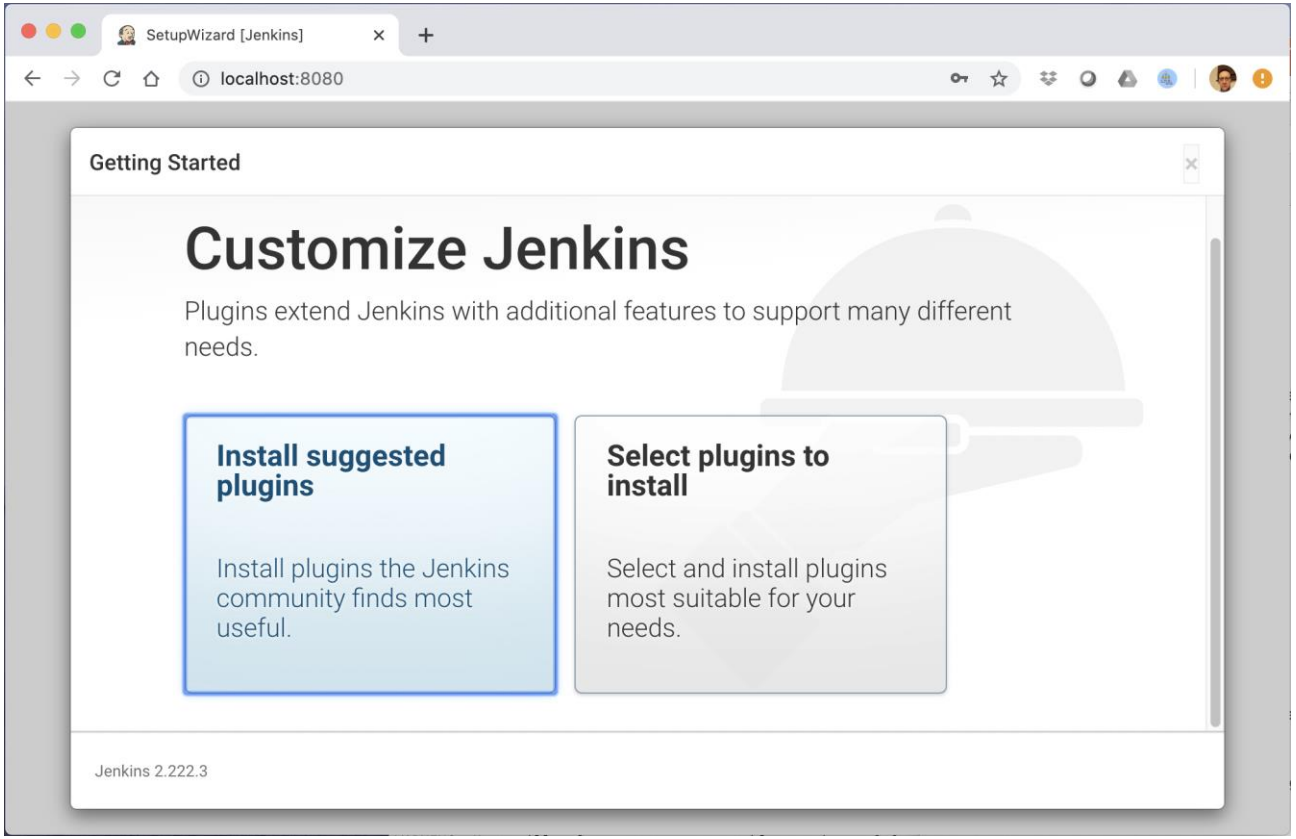
- Complete Getting Started steps to unlock Jenkins



- Administrator password in command line output

Customize Jenkins

- Can now customize Jenkins plugins



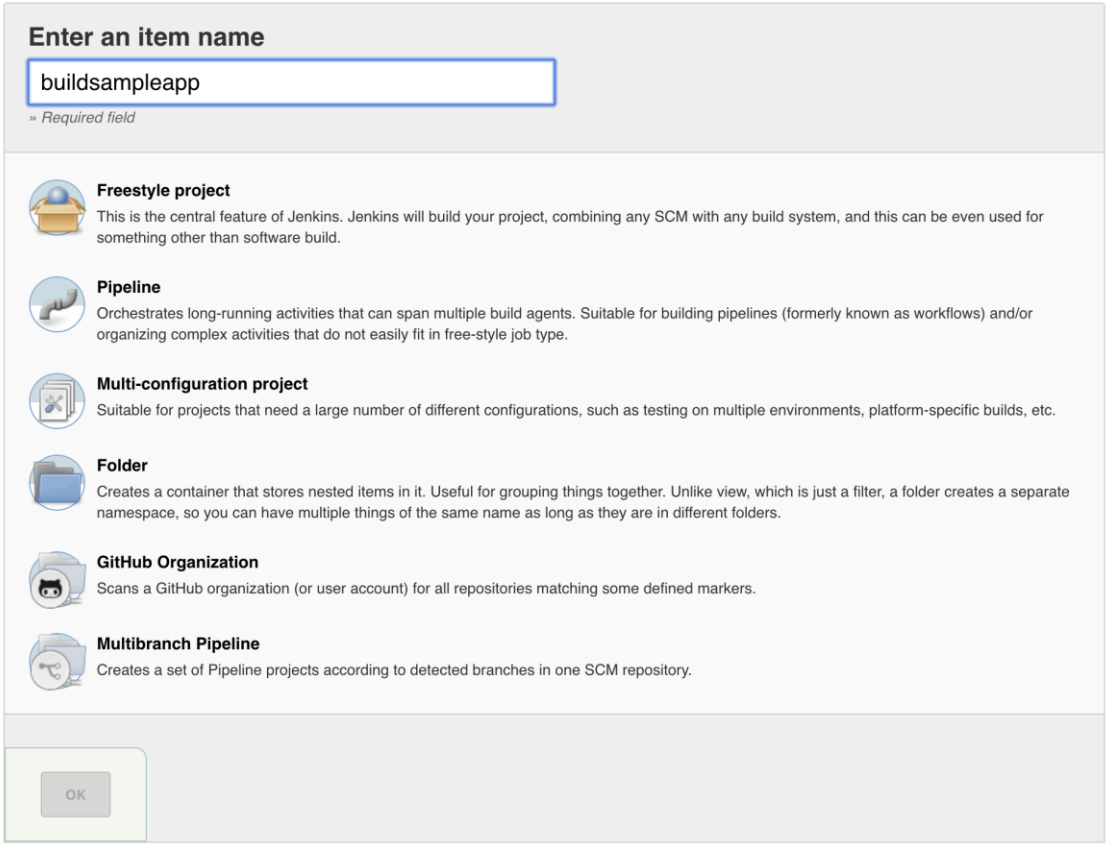
- Select suggested plugins – includes Git

Working with Jenkins

- Defines projects
  - each project has one or more jobs
- A job is a step in a build process
  - a job may involve pulling code from git
  - running Maven clean
  - running maven test
  - running Maven install
  - using docker to create an image
  - loading an image to a Docker repository
- Can define interactively or using scripting language

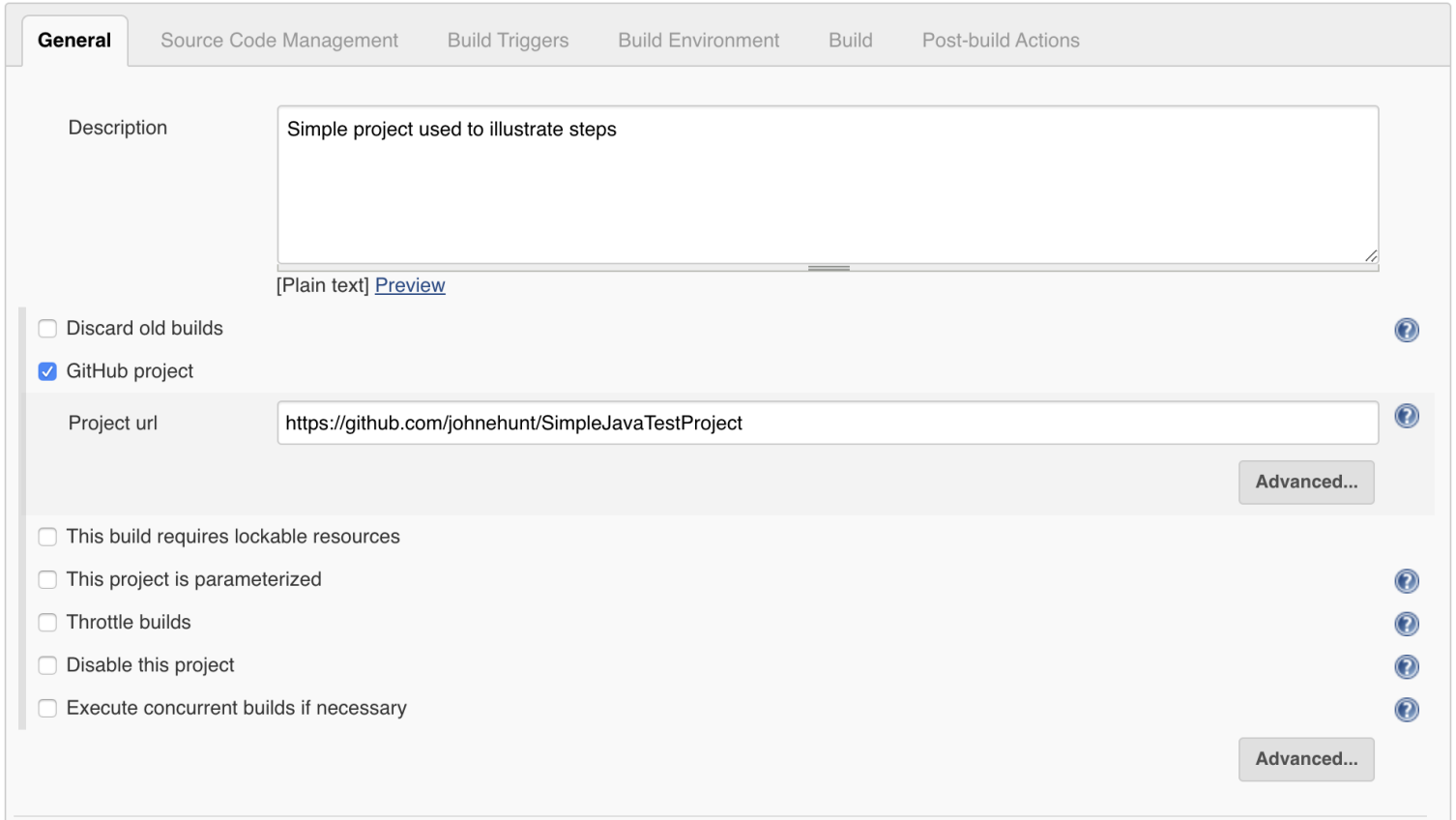
Creating a Jenkins Project

- Select New Job and select type of job e.g. Freestyle project



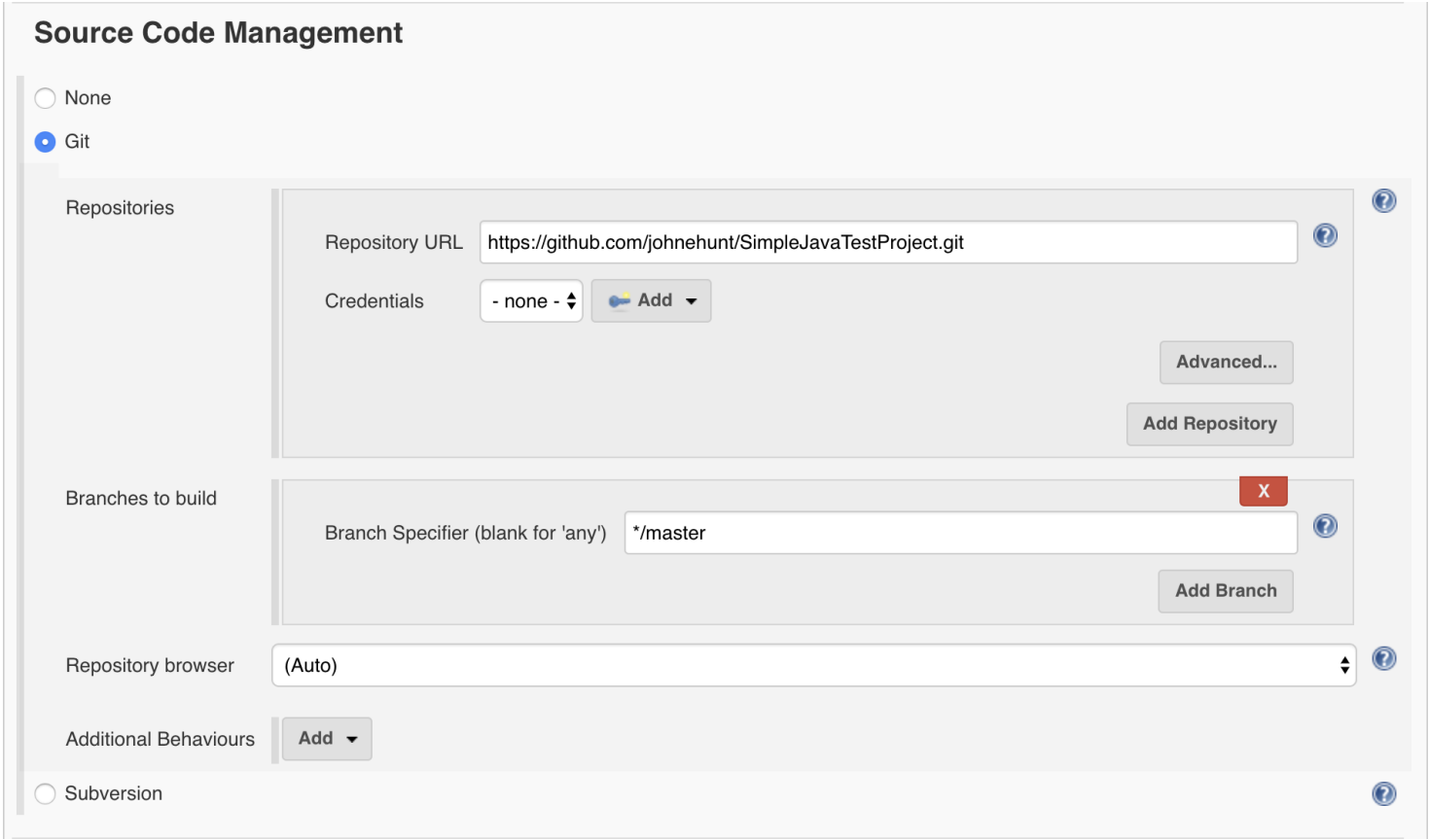
Creating a Jenkins Project

- Indicate that this is a Git project and the url of the overall project



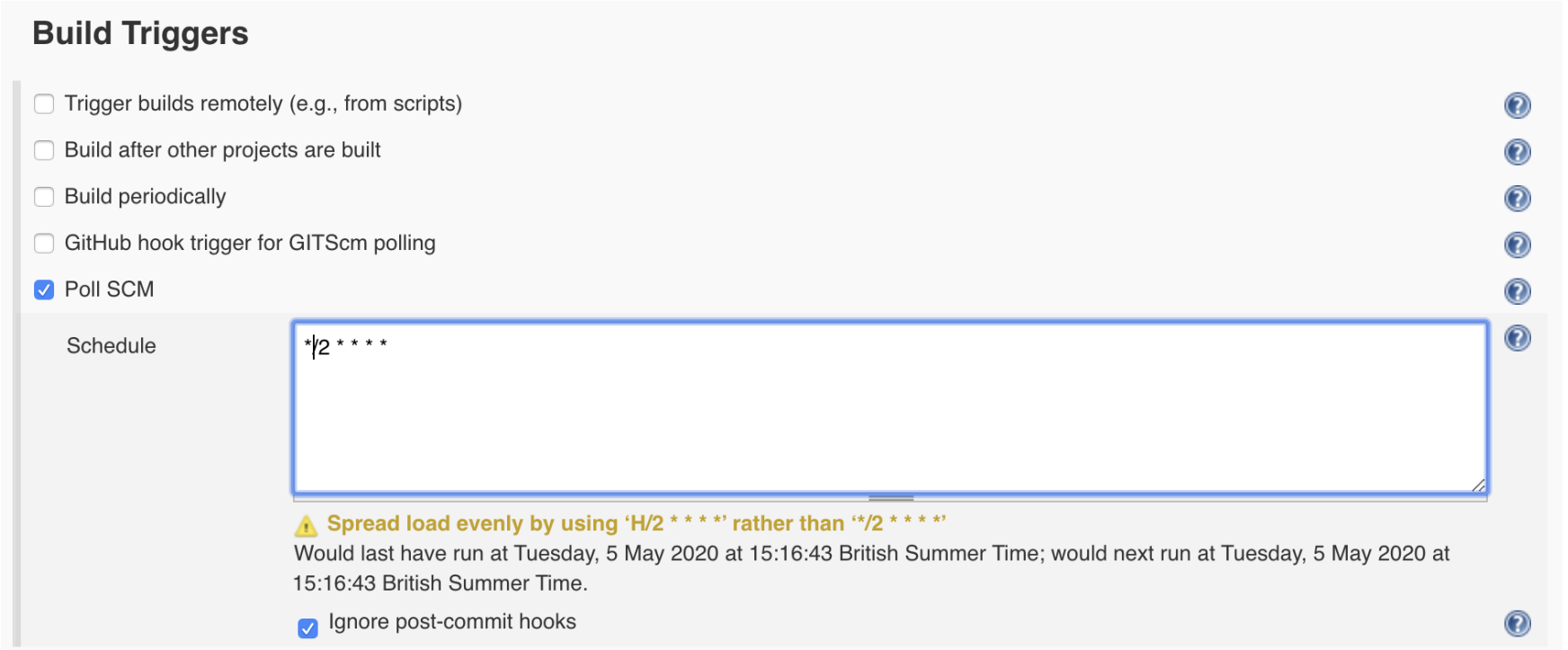
Creating a Jenkins Project

- Specify the repository and the branch to use



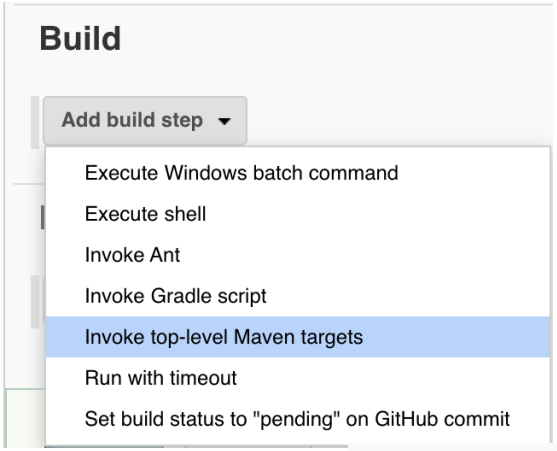
Specifying build triggers

- Determine when the build process will be triggered



Creating a Jenkins Job

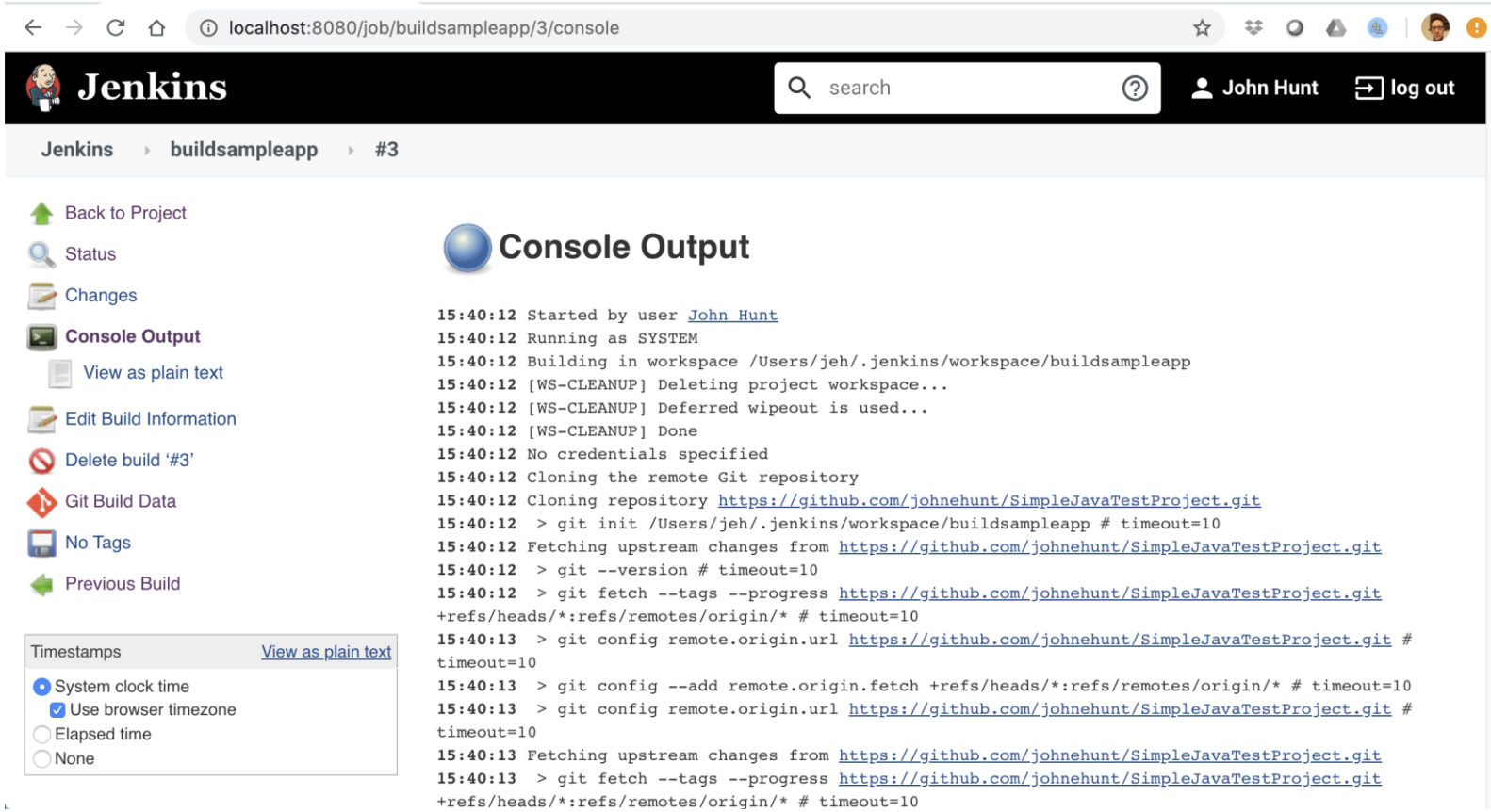
- Configure the job (build step)
  - we will use Maven for this step



can have 1 or more steps

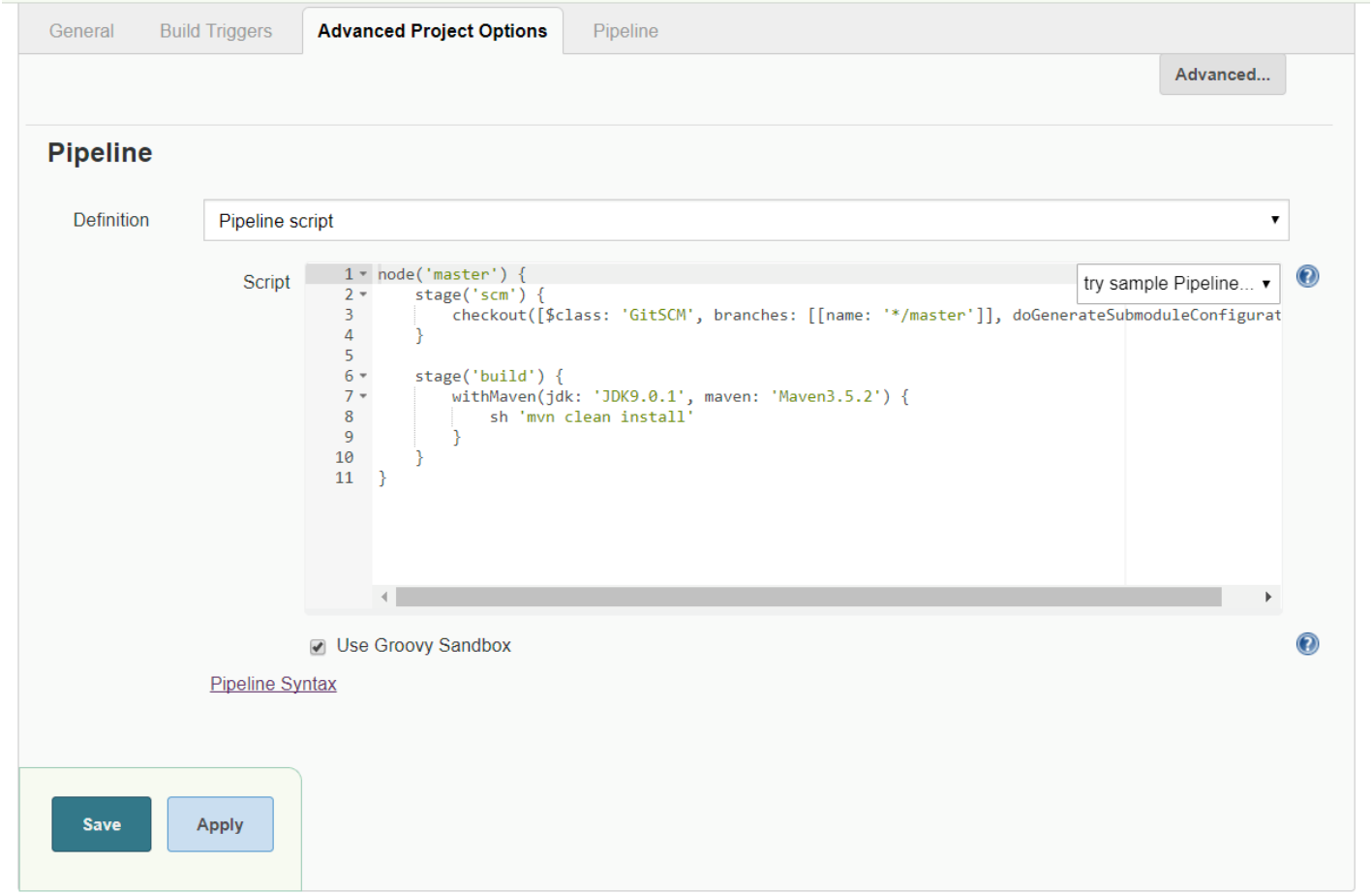
Running Jenkins Job

- Can now run the build job



Jenkins Pipelines aka Workflows

- Can be used to specify series of steps to perform



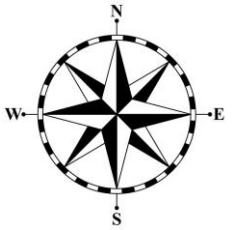
Security Landscape

Toby Dussek

Informed Academy

framework training
business value through education

Plan for Session



- Why is Security Important?
- Who are the Threats
  - Spear Fishing
- How can security be compromised?
- What can be done?
- OWASP Top Ten Vulnerabilities

Why is Security Important?

?

Why is Security Important?

- Loss to client
  - of their personal data
  - financial loss
  - time wasted / worry / upset
- Loss to Company
  - financial loss
  - reputational loss / clients leaving
  - legal / license repercussions
  - operational loss

Who are the Threats?

?

Who are the Threats?

- You are! insider Threat
- Lone Hacker – personal pride / reputation
- Cyber criminal – out for financial gain
- Hacktivist – has a specific agenda
- Nation State – politically motivated
- Investigative Hackers – journalists
- Competitors – business advantage

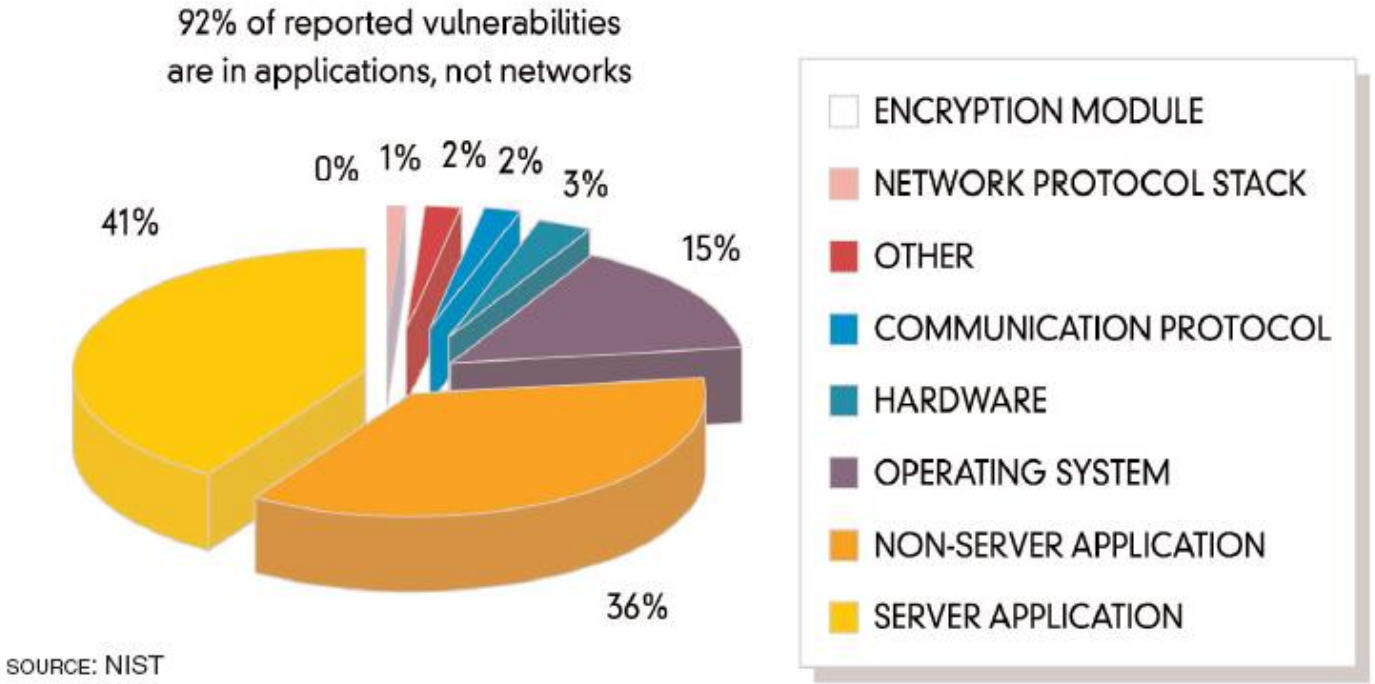
Spear Phishing

- Targets individuals
- Weakest link in the chain
  - 95% of all attacks involve people
- Try to get useful information from employees
  - may pose as colleagues needing or offering help
- May just be part of the chain
  - to get enough info to fool someone else

How can Security be Compromised?

A Few Facts and figures

How Many Vulnerabilities Are Application Security Related?

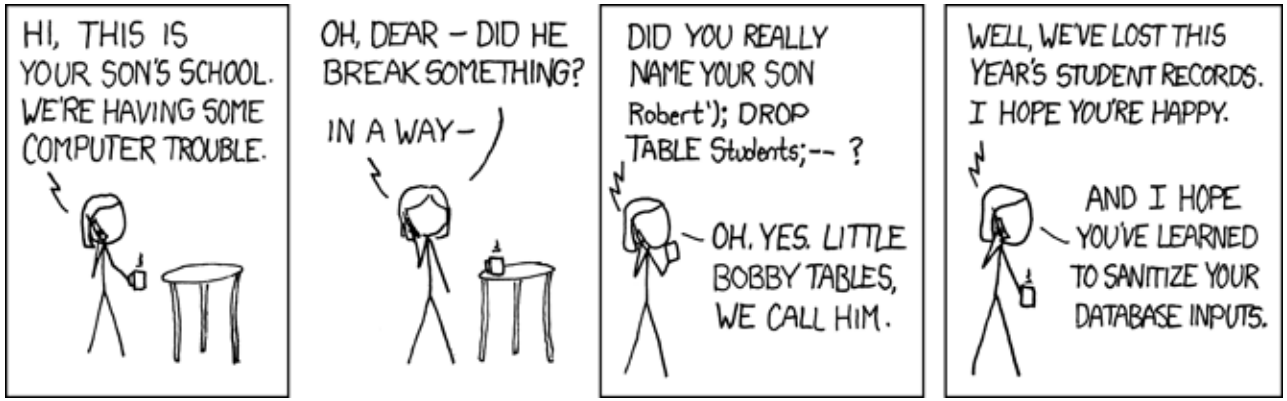


What can be Done?

- Application Security
  - common weak point
- Operating System Security
  - can be as simple as basic patching of OS
  - install antivirus / anti malware
- Hardware Security
  - encrypt drives / encrypt contents of database etc.
  - password protect files, disks, USB drives
- Network Security
  - Careful of data exchanged, emails, social media etc.
  - install firewalls, have a DMZ

What can be Done?

- Users and credentials
  - least privilege - users have no access by default
  - no users have production access
  - use different users for install and execute
- Web Applications
  - common weak spot
  - see SQL Injection example



Why are Web Apps Insecure?

- Traditional Approach
  - Crunchy Shell, Soft Centre
  - Traditional Security – perimeter Security
  - Defending the boundaries
  - Firewalls are walls
  - Ports are gateways
  - IDS (Intrusion Detection Systems) – Guards
- But the World has changed

Why are Web Apps Insecure?

- Modern (interconnected) world
  - More complex, more options
- The Web makes it different!
  - A different environment
  - Browser == Operating Systems
  - Rich Data == Code
  - Network == Computer
  - Web App Security goes across technologies / hardware / organisational boundaries

OWASP Top Ten - 2017

Attack	Description
Injection	SQL, NoSQL, OS, LDAP Injection flaws
Broken Authentication	compromises in passwords, keys or session tokens
Sensitive Data Exposure	e.g. sending PII over internet
XML External Entities	older XML processor vulnerability
Broken Access Control	Restrictions on what authenticated users are allowed to do not properly enforced.
Security Misconfiguration	which allows unauthorized access
Cross-Site Scripting	XSS for one site to another
Insecure Deserialization	can result in code execution
Using components with known vulnerabilities	such as old versions of frameworks
Insufficient Logging & Monitoring	allows attacks to continue unnoticed