# The LockManager App Case Study: Client Structure & Functionality


Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

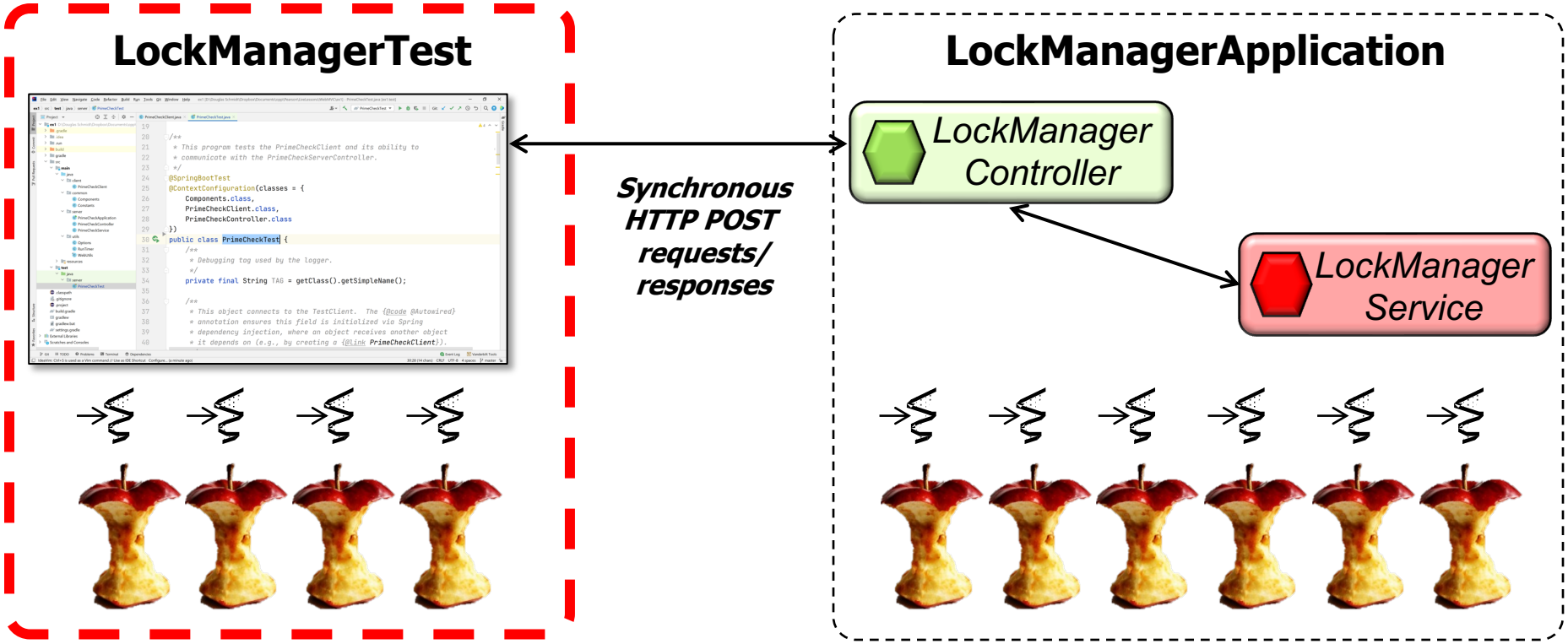
Institute for Software Integrated Systems

Vanderbilt University

Nashville, Tennessee, USA


# Learning Objectives in this Part of the Lesson

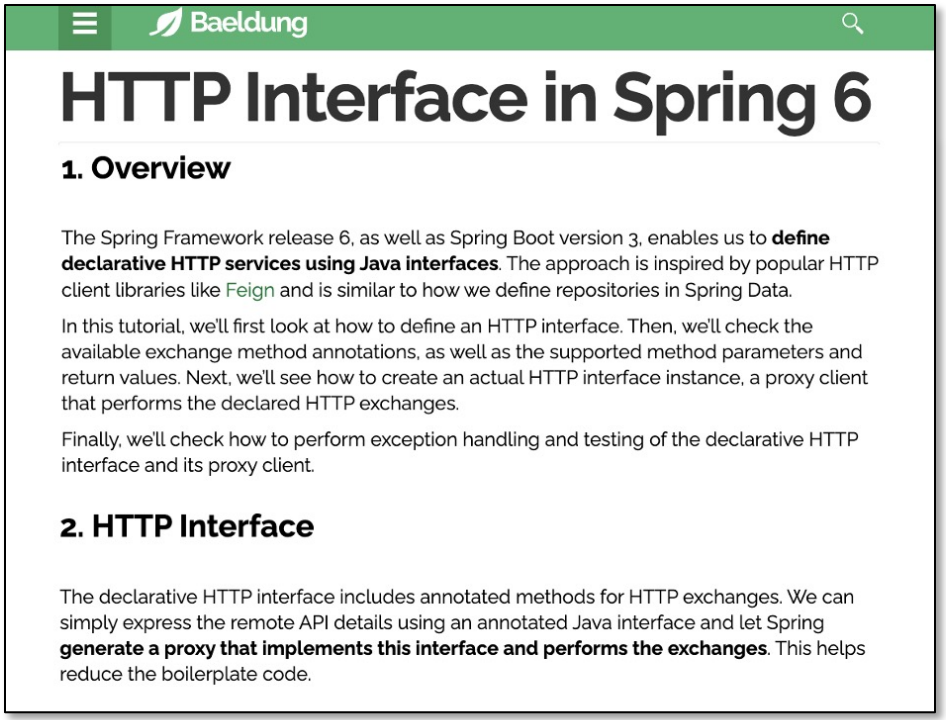
- Understand the structure & functionality of client components that send/receive HTTP POST requests/responses to/from the microservice synchronously




See github.com/douglascraigschmidt/LiveLessons/tree/master/WebMVC/ex5

# Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of client components that send/receive HTTP POST requests/responses to/from the microservice synchronously
- Recognize how the Spring HTTP Interface Clients feature works



Baeldung

## HTTP Interface in Spring 6

### 1. Overview

The Spring Framework release 6, as well as Spring Boot version 3, enables us to **define declarative HTTP services using Java interfaces**. The approach is inspired by popular HTTP client libraries like Feign and is similar to how we define repositories in Spring Data.

In this tutorial, we'll first look at how to define an HTTP interface. Then, we'll check the available exchange method annotations, as well as the supported method parameters and return values. Next, we'll see how to create an actual HTTP interface instance, a proxy client that performs the declared HTTP exchanges.

Finally, we'll check how to perform exception handling and testing of the declarative HTTP interface and its proxy client.

### 2. HTTP Interface

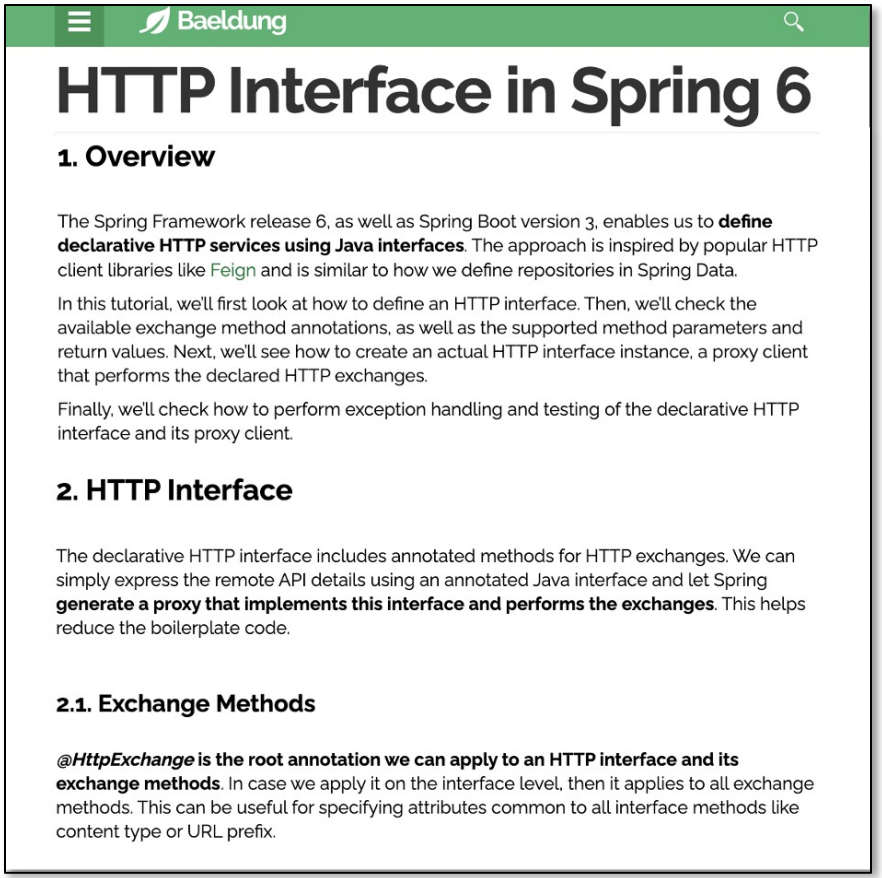
The declarative HTTP interface includes annotated methods for HTTP exchanges. We can simply express the remote API details using an annotated Java interface and let Spring **generate a proxy that implements this interface and performs the exchanges**. This helps reduce the boilerplate code.

See www.baeldung.com/spring-6-http-interface

# Overview of the HTTP Interface Clients Feature

# Overview of the HTTP Interface Clients Feature

- Spring's 6 HTTP Interface Clients feature can define client-side HTTP resources declaratively



Baeldung

## HTTP Interface in Spring 6

### 1. Overview

The Spring Framework release 6, as well as Spring Boot version 3, enables us to **define declarative HTTP services using Java interfaces**. The approach is inspired by popular HTTP client libraries like Feign and is similar to how we define repositories in Spring Data.

In this tutorial, we'll first look at how to define an HTTP interface. Then, we'll check the available exchange method annotations, as well as the supported method parameters and return values. Next, we'll see how to create an actual HTTP interface instance, a proxy client that performs the declared HTTP exchanges.

Finally, we'll check how to perform exception handling and testing of the declarative HTTP interface and its proxy client.

### 2. HTTP Interface

The declarative HTTP interface includes annotated methods for HTTP exchanges. We can simply express the remote API details using an annotated Java interface and let Spring **generate a proxy that implements this interface and performs the exchanges**. This helps reduce the boilerplate code.

#### 2.1. Exchange Methods

***@HttpExchange* is the root annotation we can apply to an HTTP interface and its exchange methods**. In case we apply it on the interface level, then it applies to all exchange methods. This can be useful for specifying attributes common to all interface methods like content type or URL prefix.

See www.baeldung.com/spring-6-http-interface

# Overview of the HTTP Interface Clients Feature

- Spring's 6 HTTP Interface Clients feature can define client-side HTTP resources declaratively
  - Declarative HTTP Clients
    - Define HTTP clients using simple Java-like interfaces

```
public interface LockAPI {
  @PostExchange(ACQUIRE_LOCKS)
  List<Lock> acquire
    (@RequestParam
     LockManager lockManager,
     @RequestParam
     Integer permits);
  ...
  @PostExchange(RELEASE_LOCK)
  Boolean release
     (@RequestParam LockManager
        lockManager,
      @RequestBody Lock lock);
  ...
}
```

# Overview of the HTTP Interface Clients Feature

- Spring's 6 HTTP Interface Clients feature can define client-side HTTP resources declaratively
  - Declarative HTTP Clients
    - Define HTTP clients using simple interfaces
  - Avoid manual creation of client code for RESTful calls

```
var uri = UriComponentsBuilder
   .fromPath(ACQUIRE_LOCKS)

   .queryParam(LOCK_MANAGER,
               lockManager)

   .queryParam(PERMITS, permits)

   .build()

   .toUriString();

...
```

# Overview of the HTTP Interface Clients Feature

- Spring's 6 HTTP Interface Clients feature can define client-side HTTP resources declaratively
  - Annotation-Driven
    - Utilize Spring MVC annotations to map client interface methods to HTTP requests

```
public interface LockAPI {
  @PostExchange(ACQUIRE_LOCK)
  Lock acquire(@RequestParam
    LockManager lockManager);

  ...
  @PostExchange(RELEASE_LOCK)
  Boolean release
    (@RequestParam LockManager
     lockManager,
     @RequestBody Lock lock);
  ...
}
```

# Overview of the HTTP Interface Clients Feature

- Spring's 6 HTTP Interface Clients feature can define client-side HTTP resources declaratively
  - Annotation-Driven
    - Utilize Spring MVC annotations to map client interface methods to HTTP requests
    - Maintain consistency with Spring's server-side annotations

```
public interface LockAPI {
   @PostExchange(ACQUIRE_LOCK)
   Lock acquire(@RequestParam
     LockManager lockManager);

   ...
   @PostExchange(RELEASE_LOCK)
   Boolean release
     (@RequestParam LockManager
      lockManager,
      @RequestBody Lock lock);
   ...
}
```

# Overview of the HTTP Interface Clients Feature

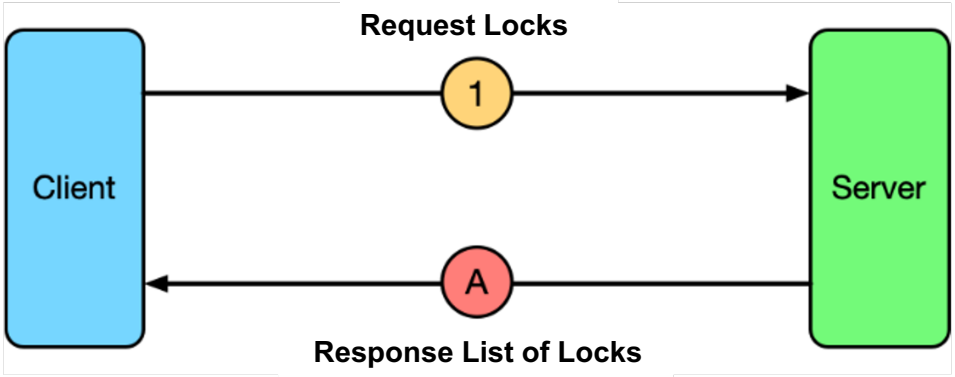
- Spring's 6 HTTP Interface Clients feature can define client-side HTTP resources declaratively
  - WebClient Integration
    - Seamlessly works with the non-blocking, reactive Web Client for HTTP calls

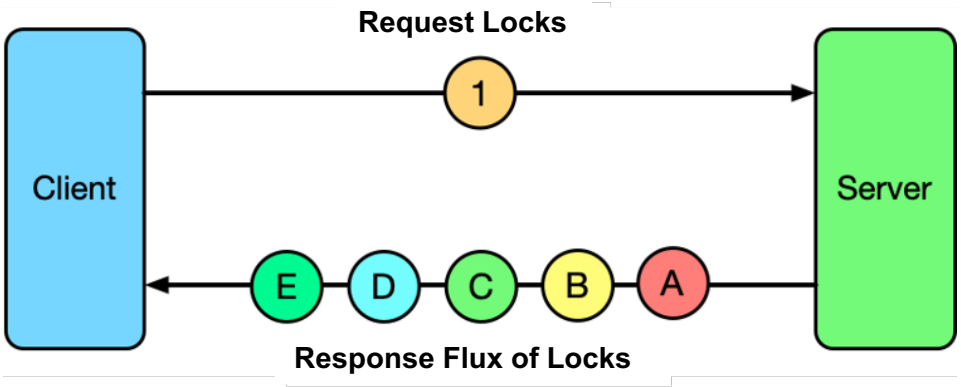
```
public LockAPI getLockAPI() {
  var webClient =
    WebClient
      .builder()
      .baseUrl(SERVER_BASE_URL)
      .build();

  ...
}
```

See www.baeldung.com/spring-5-webclient

# Overview of the HTTP Interface Clients Feature

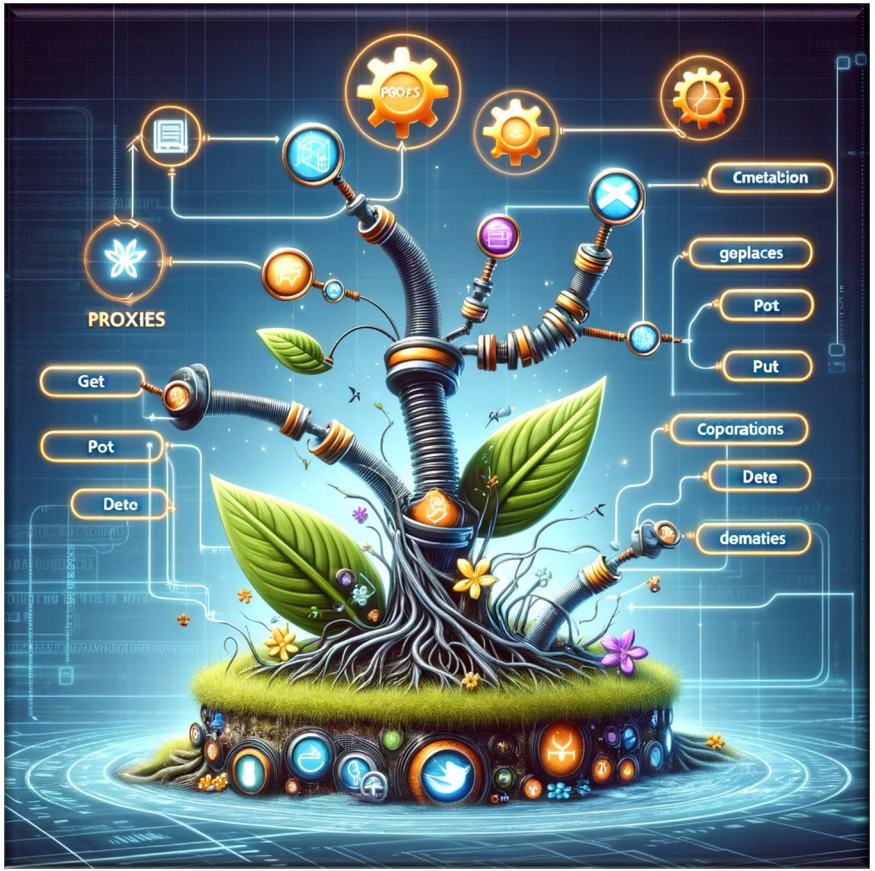
- Spring's 6 HTTP Interface Clients feature can define client-side HTTP resources declaratively
  - WebClient Integration
    - Seamlessly works with the non-blocking, reactive Web Client for HTTP calls
    - Supports both synchronous & asynchronous communication



# Overview of the HTTP Interface Clients Feature

- Spring's 6 HTTP Interface Clients feature can define client-side HTTP resources declaratively
  - Dynamic Proxy Implementation
    - Spring auto-generates proxy classes that implement an interface at runtime



# Overview of the HTTP Interface Clients Feature

- Spring's 6 HTTP Interface Clients feature can define client-side HTTP resources declaratively
  - Dynamic Proxy Implementation
    - Spring auto-generates proxy classes that implement an interface at runtime
    - Simplifies the implementation by abstracting the HTTP request handling
      - Proxies handle details of establishing HTTP connections, sending requests, receiving responses, & converting responses to Java objects

# Overview of the HTTP Interface Clients Feature

- Spring's 6 HTTP Interface Clients feature can define client-side HTTP resources declaratively
  - Type Safety
    - Compile-time checking for URL paths, query parameters, & body objects

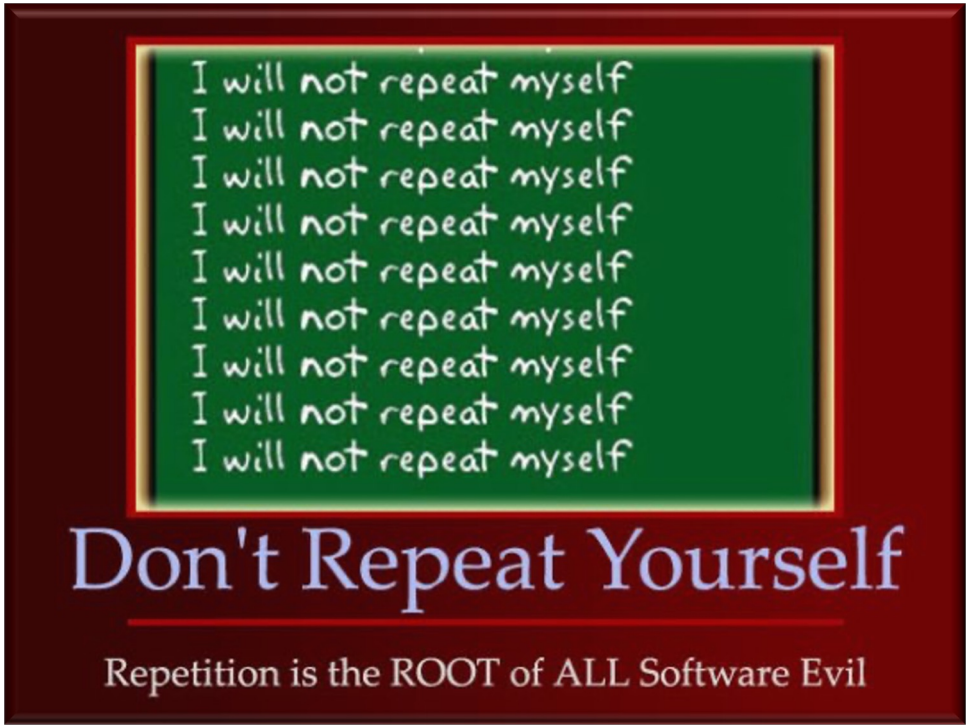
```
public interface LockAPI {
  @PostExchange(ACQUIRE_LOCK)
  Lock acquire(@RequestParam
    LockManager lockManager);

  ...
  @PostExchange(RELEASE_LOCK)
  Boolean release
    (@RequestParam LockManager
     lockManager,
     @RequestBody Lock lock);
  ...
}
```

# Overview of the HTTP Interface Clients Feature

- Spring's 6 HTTP Interface Clients feature can define client-side HTTP resources declaratively
  - Type Safety
    - Compile-time checking for URL paths, query parameters, & body objects
    - Minimizes runtime errors due to type mismatches

# Overview of the HTTP Interface Clients Feature

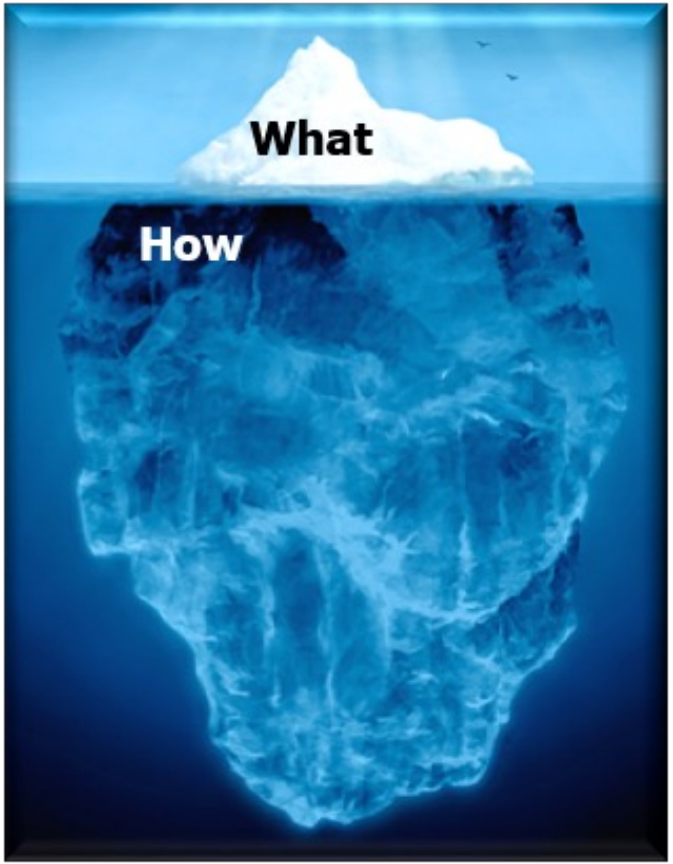
- Spring's 6 HTTP Interface Clients feature can define client-side HTTP resources declaratively
  - Reduced Boilerplate
    - Minimize repetitive code for handling HTTP connections & responses




See en.wikipedia.org/wiki/Don't_repeat_yourself

# Overview of the HTTP Interface Clients Feature

- Spring's 6 HTTP Interface Clients feature can define client-side HTTP resources declaratively
  - Reduced Boilerplate
    - Minimize repetitive code for handling HTTP connections & responses
  - Focus on defining operations, not the underlying mechanics
    - i.e., the "what" not the "how"



# The Structure & Functionality of LockAPI Interface

# The Structure & Functionality of the LockAPI Interface

- The LockAPI interface hides details of remote method invocations via HTTP

```
public interface LockAPI {
   @PostExchange(CREATE)
   LockManager create(@RequestParam

   @PostExchange(ACQUIRE_LOCK)
   Lock acquire(@RequestParam LockManager lockManager);

   ...
   @PostExchange(RELEASE_LOCK)
   Boolean release(@RequestParam LockManager lockManager,
                   @RequestBody Lock lock);
   ...
}
```

*This design uses the declarative Spring 6 HTTP interface features*

See www.baeldung.com/spring-6-http-interface

# The Structure & Functionality of the LockAPI Interface

- The LockAPI interface hides details of remote method invocations via HTTP

```
public interface LockAPI {
   @PostExchange(CREATE)
   LockManager create(@RequestParam Integer maxLocks);

   @PostExchange(ACQUIRE_LOCK)
   Lock acquire(@RequestParam LockManager

   ...
   @PostExchange(RELEASE_LOCK)
   Boolean release(@RequestParam LockManager lockManager,
                   @RequestBody Lock lock);
   ...
}
```

*These proxy methods shield clients from low-level details of HTTP programming*

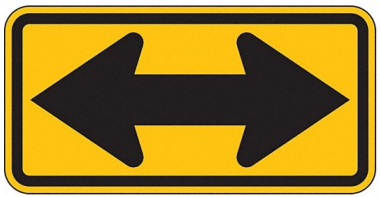
Spring 6 HTTP interface features are cleaner than Retrofit, but less pervasive

# The Structure & Functionality of the LockAPI Interface

- The LockAPI interface hides details of remote method invocations via HTTP

```
public interface LockAPI {
   @PostExchange(CREATE)
   LockManager create(@RequestParam Integer maxLocks);

   @PostExchange(ACQUIRE_LOCK)
   Lock acquire(@RequestParam LockManager lockManager);

   ...
   @PostExchange(RELEASE_LOCK)
   Boolean release(@RequestParam LockManager lockManager,
                   @RequestBody Lock lock);
   ...
}
```

*These two-way calls are all synchronous & they return conventional Java types*



# The Structure & Functionality of the LockAPI Interface

- The LockAPI interface hides details of remote method invocations via HTTP

```
public interface LockAPI {
   @PostExchange(CREATE)
   LockManager create(@RequestParam Integer maxLocks);

   @PostExchange(ACQUIRE_LOCK)
   Lock acquire(@RequestParam LockManager lockManager);

   ...
   @PostExchange(RELEASE_LOCK)
   Boolean release(@RequestParam LockManager lockManager,
                   @RequestBody Lock lock);
   ...
}
```

*These annotations mark these methods as HTTP POST endpoints*

See http-declarative-http-client-httpexchange/#3-creating-an-http-service-interface

# The Structure & Functionality of the LockAPI Interface

- The LockAPI interface hides details of remote method invocations via HTTP

```
public interface LockAPI {
   @PostExchange(CREATE)
   LockManager create(@RequestParam Integer maxLocks);

   @PostExchange(ACQUIRE_LOCK)
   Lock acquire(@RequestParam LockManager lockManager);

   ...
   @PostExchange(RELEASE_LOCK)
   Boolean release(@RequestParam LockManager lockManager,
                   @RequestBody Lock lock);
   ...
}
```

*These paths identify a specific HTTP endpoint*

# The Structure & Functionality of the LockAPI Interface

- The LockAPI interface hides details of remote method invocations via HTTP

```
public interface LockAPI {
   @PostExchange(CREATE)
   LockManager create(@RequestParam Integer maxLocks);

   @PostExchange(ACQUIRE_LOCK)
   Lock acquire(@RequestParam LockManager lockManager);

   ...
   @PostExchange(RELEASE_LOCK)
   Boolean release(@RequestParam LockManager lockManager,
                   @RequestBody Lock lock);
   ...
}
```

*These annotations are the same ones used by a Spring controller*

See www.java67.com/2023/02/requestparam-vs-requestbody-in-spring.html

# Creating an Instance of the LockAPI Interface

# Creating an Instance of the LockAPI Interface

- The ClientBeans class contains a factory method bean that creates the LockAPI proxy that uses the Spring 6 HTTP interface features

```
@Component
public class ClientBeans {
  @Bean
  public LockAPI getLockAPI() {
    var webClient = WebClient.builder()
        .baseUrl(LOCK_MANAGER_SERVER_BASE_URL).build();

    return HttpServiceProxyFactory
        .builder(WebClientAdapter
                 .forClient(webClient))
        .blockTimeout(Duration.ofSeconds(sTIMEOUT_DURATION))
        .build()
        .createClient(LockAPI.class); ...
```

See WebMVC/ex5/src/test/java/edu/vandy/lockmanager/ClientBeans.java

# Creating an Instance of the LockAPI Interface

- The ClientBeans class contains a factory method bean that creates the LockAPI proxy that uses the Spring 6 HTTP interface features

```
@Component
public class ClientBeans {
  @Bean
  public LockAPI getLockAPI() {
    var webClient = WebClient.builder()
        .baseUrl(LOCK_MANAGER_SERVER_BASE_URL).build();

    return HttpServiceProxyFactory
        .builder(WebClientAdapter
                 .forClient(webClient))
        .blockTimeout(Duration.ofSeconds(sTIMEOUT_DURATION))
        .build()
        .createClient(LockAPI.class); ...
```

*This @Bean annotation can be injected into classes using Spring's @Autowired annotation*

# Creating an Instance of the LockAPI Interface

- The ClientBeans class contains a factory method bean that creates the LockAPI proxy that uses the Spring 6 HTTP interface features

```
@Component
public class ClientBeans {
  @Bean
  public LockAPI getLockAPI() {
    var webClient = WebClient.builder()
        .baseUrl(LOCK_MANAGER_SERVER_BASE_URL).build();

    return HttpServiceProxyFactory
        .builder(WebClientAdapter
                 .forClient(webClient))
        .blockTimeout(Duration.ofSeconds(sTIMEOUT_DURATION))
        .build()
        .createClient(LockAPI.class); ...
```

*Create the main entry point for performing web requests (for both sync & async calls)*

See www.baeldung.com/spring-5-webclient

# Creating an Instance of the LockAPI Interface

- The ClientBeans class contains a factory method bean that creates the LockAPI proxy that uses the Spring 6 HTTP interface features

```
@Component
public class ClientBeans {
  @Bean
  public LockAPI getLockAPI() {
    var webClient = WebClient.builder()
        .baseUrl(LOCK_MANAGER_SERVER_BASE_URL).build();

    return HttpServiceProxyFactory
        .builder(WebClientAdapter
                 .forClient(webClient))
        .blockTimeout(Duration.ofSeconds(sTIMEOUT_DURATION))
        .build()
        .createClient(LockAPI.class); ...
```

*Adapt WebClient to provide a synchronous proxy using the Spring HTTP interface*

See www.baeldung.com/spring-6-http-interface

# Creating an Instance of the LockAPI Interface

- The ClientBeans class contains a factory method bean that creates the LockAPI proxy that uses the Spring 6 HTTP interface features

```
@Component
public class ClientBeans {
  @Bean
  public LockAPI getLockAPI() {
    var webClient = WebClient.builder()
        .baseUrl(LOCK_MANAGER_SERVER_BASE_URL).build();

    return HttpServiceProxyFactory
        .builder(WebClientAdapter
                 .forClient(webClient))
        .blockTimeout(Duration.ofSeconds(sTIMEOUT_DURATION))
        .build()
        .createClient(LockAPI.class); ...
```

*Extend default client timeout period*

# Creating an Instance of the LockAPI Interface

- The ClientBeans class contains a factory method bean that creates the LockAPI proxy that uses the Spring 6 HTTP interface features

```
@Component
public class ClientBeans {
  @Bean
  public LockAPI getLockAPI() {
    var webClient = WebClient.builder()
        .baseUrl(LOCK_MANAGER_SERVER_BASE_URL).build();

    return HttpServiceProxyFactory
        .builder(WebClientAdapter
                 .forClient(webClient))
        .blockTimeout(Duration.ofSeconds(sTIMEOUT_DURATION))
        .build()
        .createClient(LockAPI.class); ...
```



*Create an instance of LockAPI*

# End of the LockManager App Case Study: Client Structure & Functionality