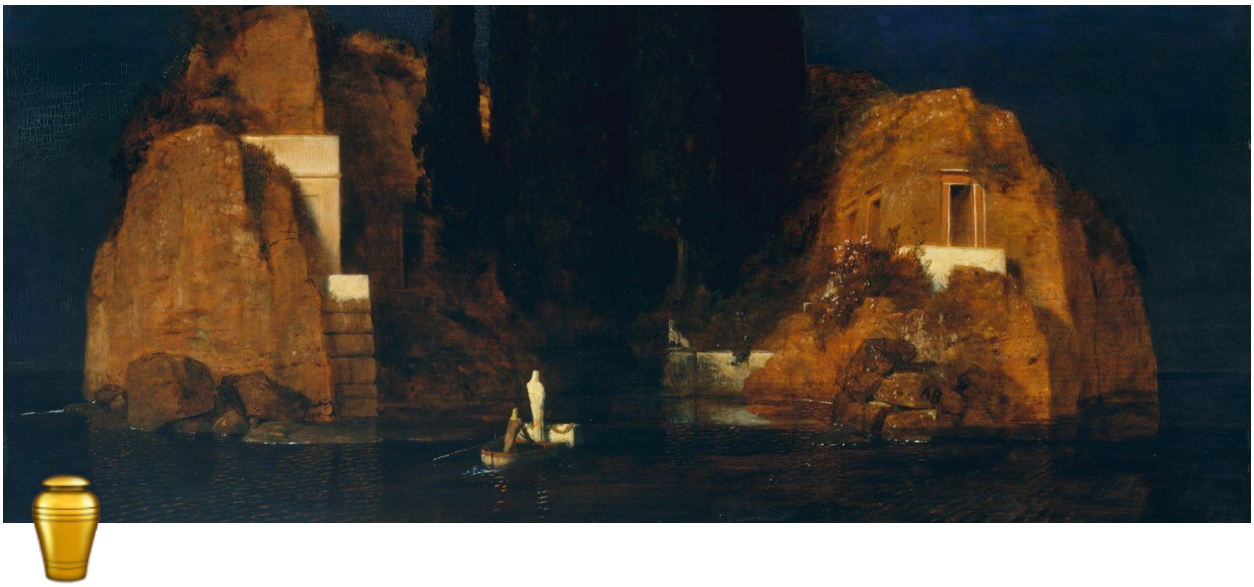
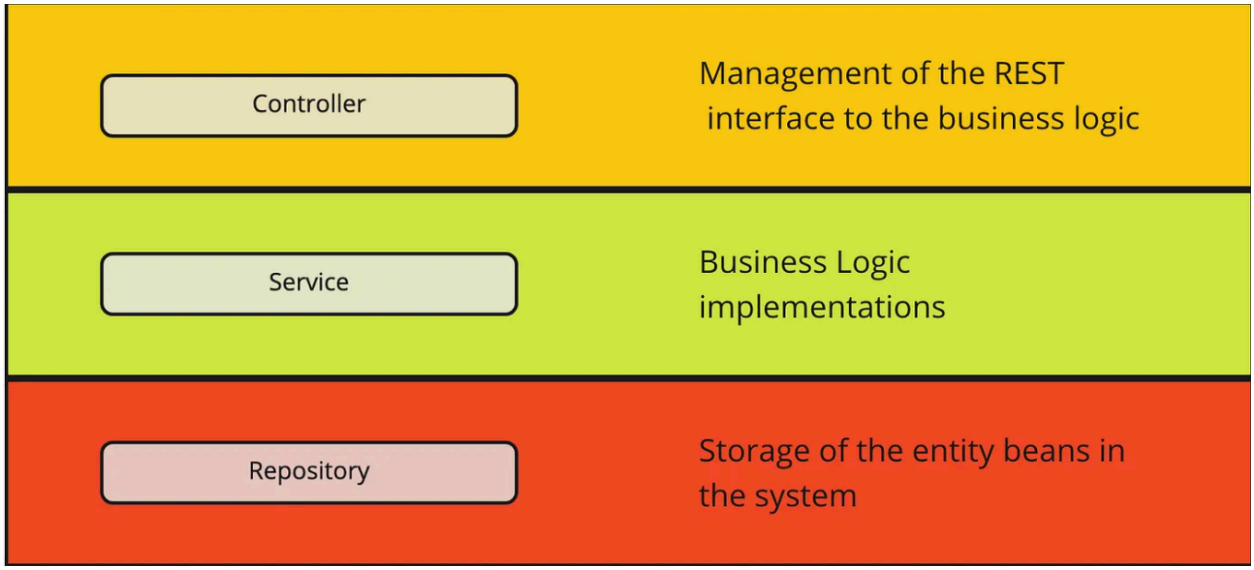


Giftwrapping Intern Assignment



Controller-Service-Repository

- **I like this pattern is that it does a great job of a separation of concerns**

- The Controller layer - Exposing the functionality so that it can be consumed by external entities
- The Repository layer - Responsible for storing and retrieving some set of data.
- The Service layer - Where all the business logic should go

```
@Getter
@Setter
@With
@NoArgsConstructor
@AllArgsConstructor
@Entity

public class Bookmark{

@Id
private long id;
private String url;

}
```

- We will now write testCase for the BookmakrController

```
@ExtendWith(SpringExtension.class)
@WebMvcTest(BookmarkController.class)
public class BookmarkControllerTest {
@MockBean
BookmarkService bookmarkService;
@AutoWired
MockMvc mockMvc;

@Test
   public void testGetById() throws Exception{
      Bookmark bookmark = new Bookmark.withId(1).withUrl("facebook.com");
```

```
        when(bookmarkService.findById(1)).thenReturn(url);
        ResultAction result = mockMvc.perform(get("/api/bookmark/1"));
        .andExpect(status().isOk())
        .andExpect(jsonPath("$url").value("facebook.com"));
        verify(bookmarkService).findById(1);

    }

}
```

- Components of the test case
    1. Mock Bookmark `Service` is injected into Bookmark `Controller`.
    2. A `GET` request to `/api/url/1` is sent via `MockMvc`.
    3. The mock service returns the predefined Bookmark object.
    4. The response is validated for status and content.
    5. The test verifies the mock interaction.
- Failing Test case let's write the controller class now

```
@RestController
public class BookmarkController{

BookmarkService bookmarkService;

public class BookmarkController(BookmarkService bookmarkService){
    this.bookmarkServer = bookmarkService;
}

@GetMapping("api/url/{id}")
public Bookmark getBookmarkById(@PathVariable long id) throws Exception {
    return bookmarkService.findById(id);
}
```

- Components of the Controller
  - `@RestController`
    - Marks this class as a **Spring MVC Controller** that handles HTTP requests.
    - Combines `@Controller` and `@ResponseBody`, meaning all methods return data (JSON/XML) instead of a view.
  - `@GetMapping("api/url/{id}")`
    - Maps HTTP `GET` requests to `/api/url/{id}` to the `getUrlById()` method.
    - `{id}` is a **path variable** (extracted from the URL).
  - `@PathVariable long id`
    - Extracts the `{id}` value from the URL and passes it as a `long` parameter to `getUrlById()`.
- Writing the testCase for the UrlService

```
@ExtendWith(MockitoExtension.class)

public class BookmarkServiceTest(){
   @Mock
   BookmarkRepository bookmarkRepo;
   @Test
   public void getBookmarkById() throws exception {
      BookmarkService bookmarkService = new BookmarkService(bookmarkRep
      Optional<Bookmark> bookmark = Optional.of(new Bookmark().withId(1).witl
      when(bookmarkRepo.findById(1)).thenReturn(bookmark);
      Bookmark foundBookmark = bookmarkService.findById(1);
      assertThat(foundBookmark).isEqualTo(bookmark.get());
      verify(bookmarkRepo).findById(1);
   }
}
```

- `Optional` is a **safer alternative to** `null`.

- handle cases where a value may or may not be present
    - Better than throwing nullpoint error, handles those cases gracefully
- Writing the Service

```
@Service
public class BookmarkService{
    private BookmarkRepository bookmarkRepository;
    public class BookmarkService(BookmarkRepository bookmarkRepository){
        this.bookmarkRepository = bookmarkRepository;
    }
    public Bookmark findById(long id) throws BookmarkNotFoundException{
        Optional<Bookmark> oBookmark = bookmarkRepository.findById(id);
        return oBookmark.get();
    }
}
```