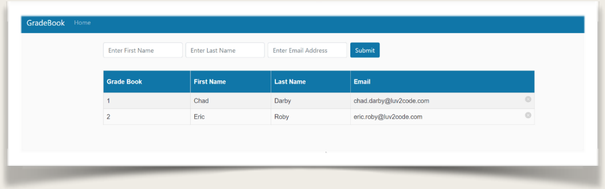
luv2code

Course Project - Testing Overview

Student Grade Book App

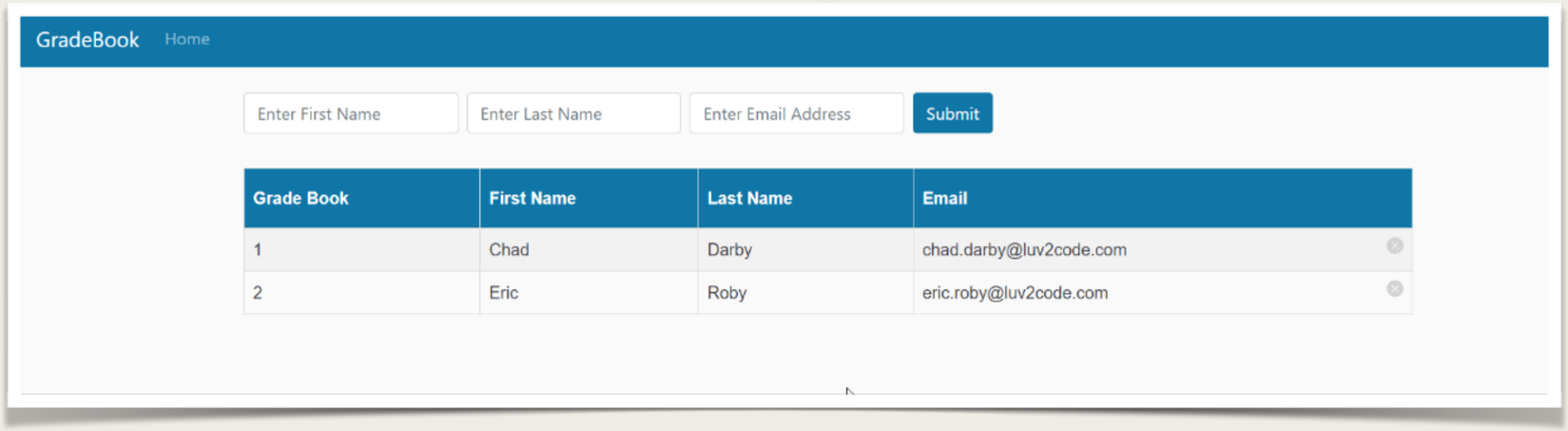
- We will start with an existing Student Grade Book App
- The app was created by a previous employee ... but it is unfinished (yikes!)



- Our job:
  - Add remaining functionality to save data in database
  - Add unit tests and integration tests

About Student Grade Book App

- An instructor can keep track of grades for a student
- Grades are tracked for the subjects: History, Science and Math
- Instructor can add grades for a student for a specific subject



Technical Stack

- Spring Boot
- Spring Data JPA
- Spring MVC
- Thymeleaf views
- CSS and JavaScript

DEMO

Existing Code

Controller

```
GradeBookController.java
```

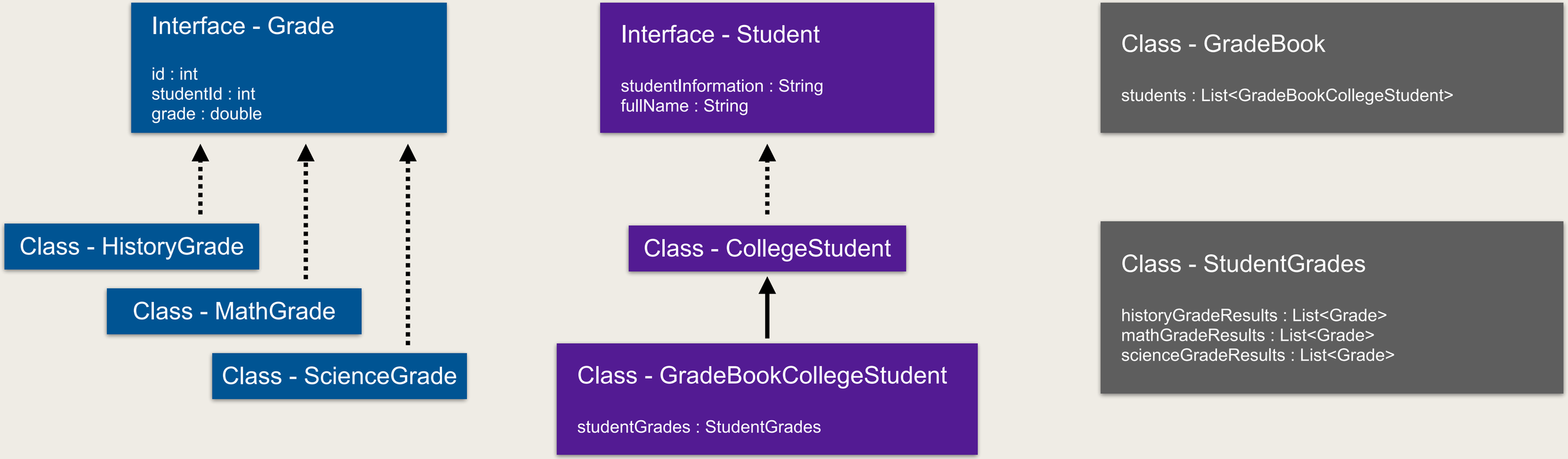
View

```
index.html
studentInformation.html
error.html
cssandjs/
```

Model

```
CollegeStudent.java
Grade.java
Gradebook.java
GradebookCollegeStudent.java
HistoryGrade.java
MathGrade.java
ScienceGrade.java
Student.java
StudentGrades.java
```

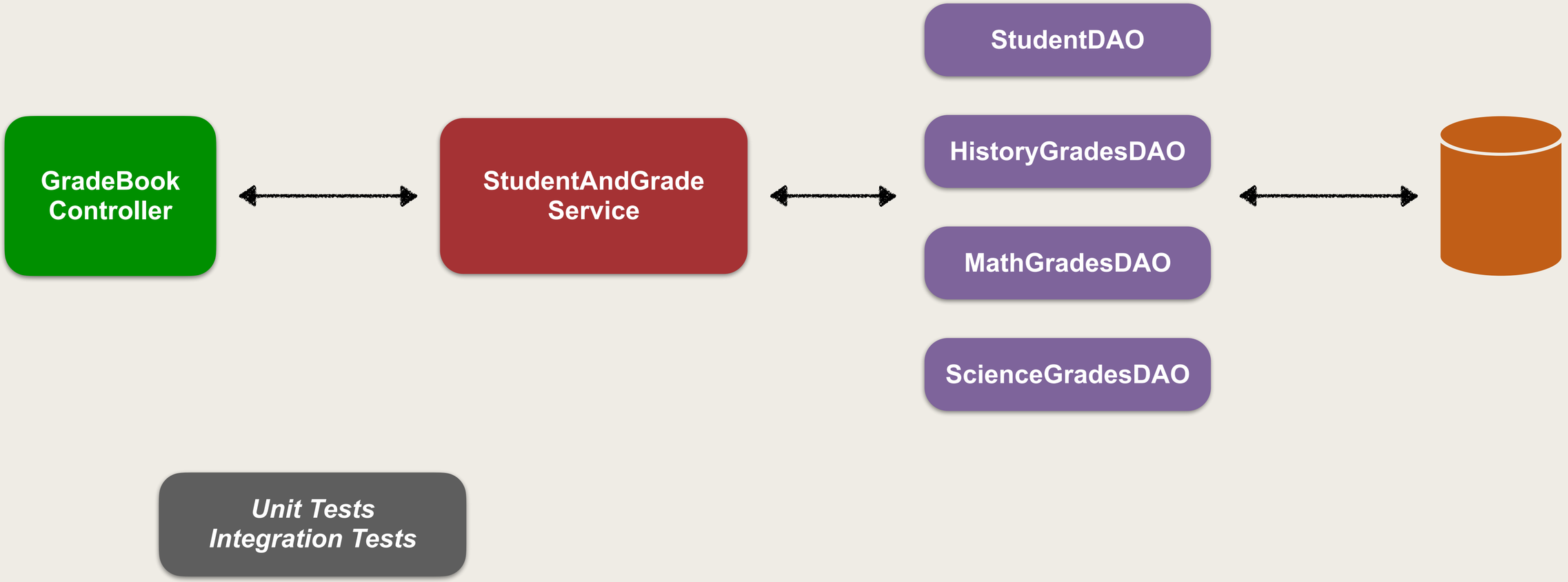
Existing Code - Model Classes



Code we will develop

- Currently, the app does not store information in database
- We'll add DAO database support
- We'll also add a service class
- During development, add unit tests and integration tests

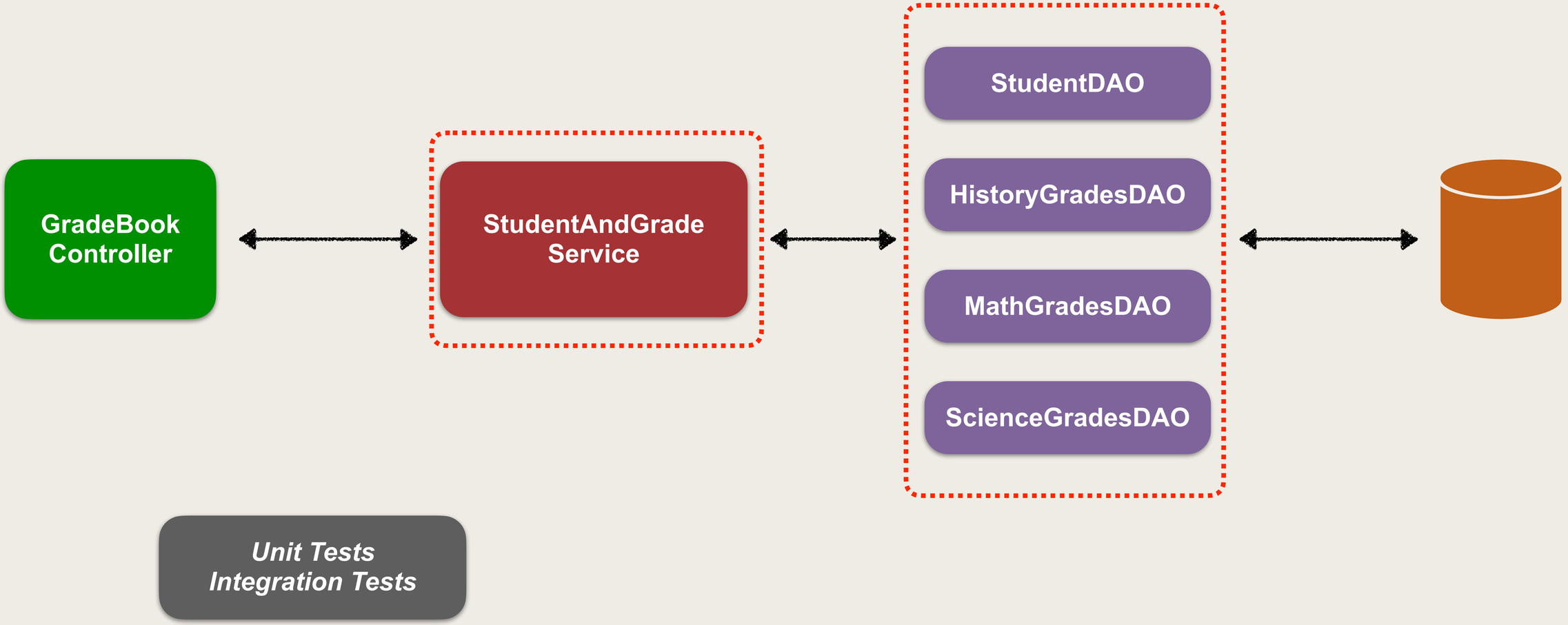
Final Architecture



luv2code

TDD for Service and DAO

Use TDD to Build Service and DAOs



DAOs and DB

- For DAOs, we will make use of Spring Data JPA
- For database, we will use H2 database (in-memory, embedded db)
  - In-memory, embedded db is good for testing
  - Quickly set up and tear down
  - No network latency so tests run faster
  - Minimizes left over data in the database

luv2code

Database Integration Testing

Database Initialization and Cleanup

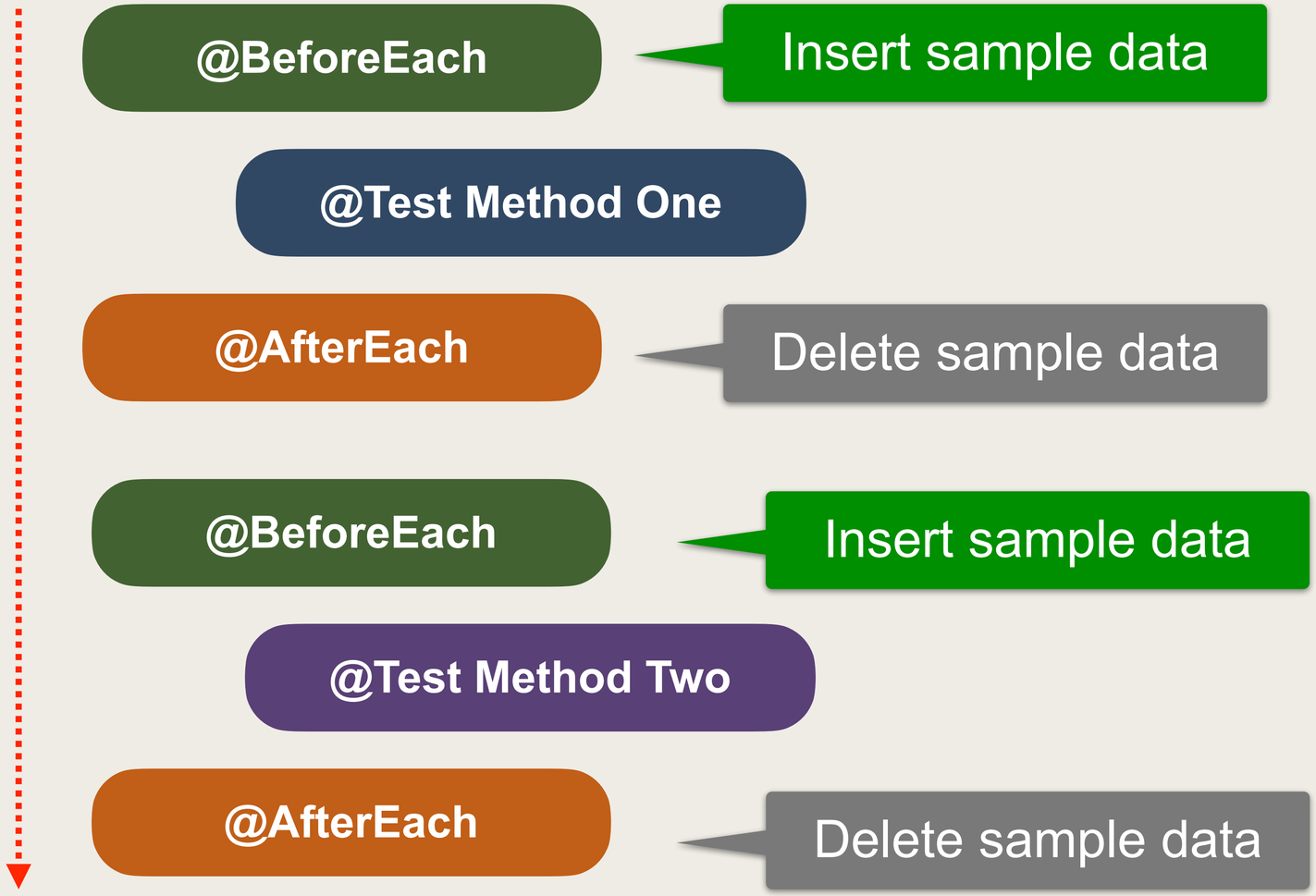
- When we are performing integration testing with a database
  - Each test should run from a known state

- Before each test, perform initialization
  - Insert sample data

- After each test, perform cleanup
  - Delete the sample data

Testing Approach

- Each test should run from a known state



@Before and @AfterEach

StudentAndGradeServiceTest.java

```
import org.springframework.jdbc.core.JdbcTemplate;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
...

@TestPropertySource("/application.properties")
@SpringBootTest
public class StudentAndGradeServiceTest {

   @Autowired
   private JdbcTemplate jdbc;

   @BeforeEach
   public void setupDatabase() {
       jdbc.execute("insert into student(id, firstname, lastname, email_address) " +
               "values (1, 'Eric', 'Roby', 'eric.roby@luv2code_school.com')");
   }


   @AfterEach
   public void setupAfterTransaction() {
       jdbc.execute("DELETE FROM student");
   }

}
```

From the Spring Framework

Insert sample data

Delete sample data

StudentAndGradeServiceTest.java

```
...
public class StudentAndGradeServiceTest {

   @Autowired
   private JdbcTemplate jdbc;

   @BeforeEach
   public void setupDatabase() {
       jdbc.execute("insert into student(id, firstname, lastname, email_address) " +
               "values (1, 'Eric', 'Roby', 'eric.roby@luv2code_school.com')");
   }

   @Test
   public void isStudentNullCheck() {

       assertTrue(studentService.checkIfStudentIsNull(1));

       assertFalse(studentService.checkIfStudentIsNull(0));
   }

   @AfterEach
   public void setupAfterTransaction() {
       jdbc.execute("DELETE FROM student");
   }

}
```

Returns true since
id 1 exists in database

Returns false since
id 0 does not exist in database

luv2code

Testing Spring MVC Web Controllers

Problem

- How can we test Spring MVC Web Controllers?
- How can we create HTTP requests and send to the controller?
- How can we verify HTTP response?
  - status code
  - view name
  - model attributes

Spring Testing Support

- Mock object support for web, REST APIs etc ...
- For testing controllers, you can use `MockMvc`
- Provides Spring MVC processing of request / response
- There is no need to run a server (embedded or external)

Development Process

1. Add annotation `@AutoConfigureMockMvc`
2. Inject the `MockMvc`
3. Perform web requests
4. Define expectations
5. Assert results

GradebookController.java

```
@Controller
public class GradebookController {

  @RequestMapping(value = "/", method = RequestMethod.GET)
  public String getStudents(Model m) {
    return "index";
  }

}
```

GradebookControllerTest.java

```
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.MvcResult;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
...

@AutoConfigureMockMvc
@SpringBootTest
public class GradebookControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void getStudentsHttpRequest () throws Exception {

        MvcResult mvcResult = mockMvc.perform(MockMvcRequestBuilders.get("/"))
                .andExpect(status().isOk()).andReturn();

        ModelAndView mav = mvcResult.getModelAndView();

        ModelAndViewAssert.assertViewName(mav, "index");

    }

}
```

Step 1: Autoconfigure

Step 2: Inject the MockMvc

Step3: Perform web requests

Step 4: Define expectations

Step 5: Assert results

Can also assert model attributes.
Retrieve model attribute objects for fine-grained asserts

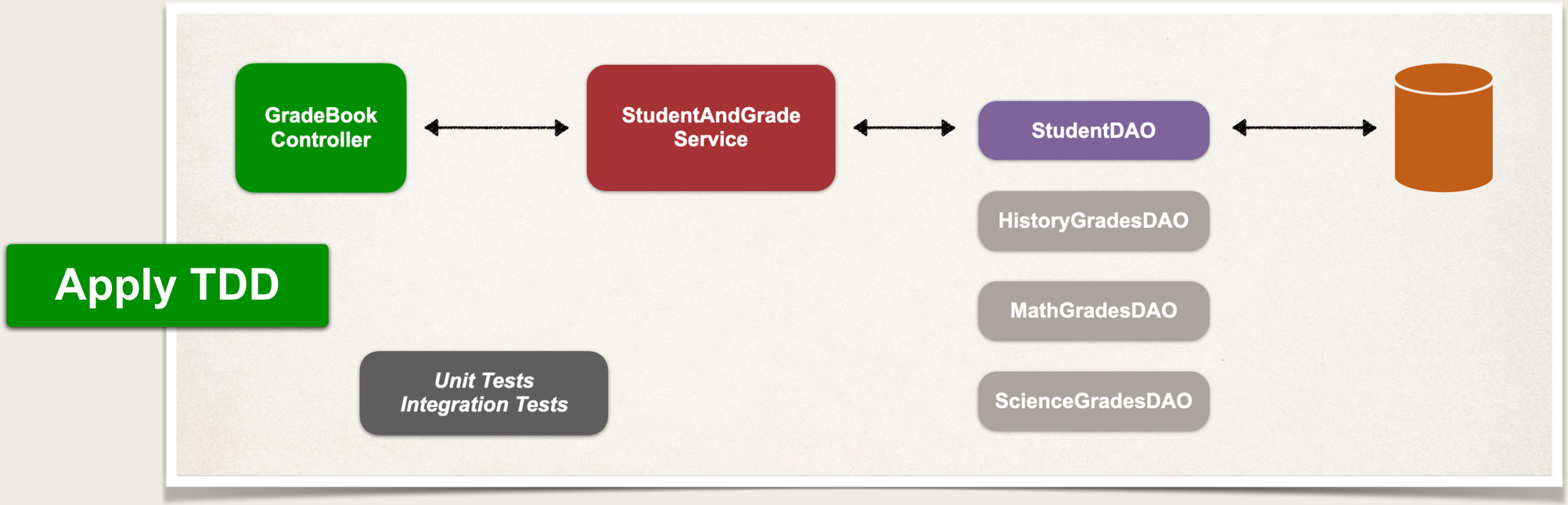
GradebookController.java

```
@Controller
public class GradebookController {

  @RequestMapping(value = "/", method = RequestMethod.GET)
  public String getStudents(Model m) {
    return "index";
  }

}
```

Test - Create Student

Test Case: Create a student in the database

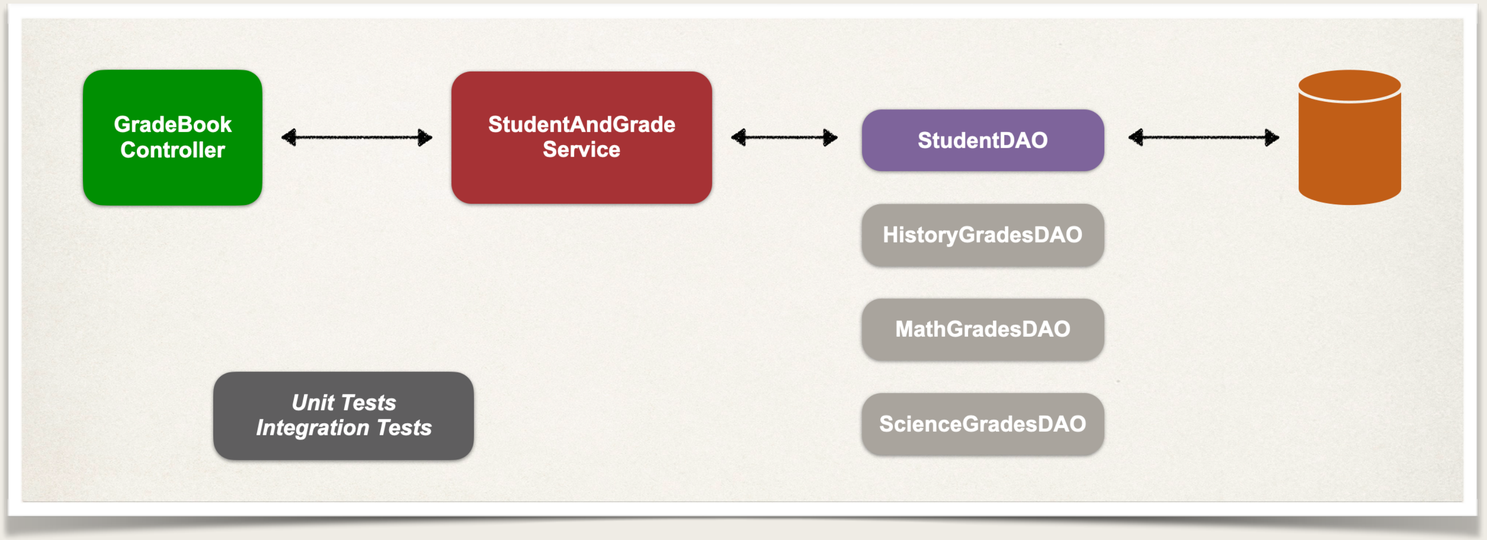
- Send a POST request to the controller
- Verify results by accessing data using the DAO



Updates for Gradebook UI

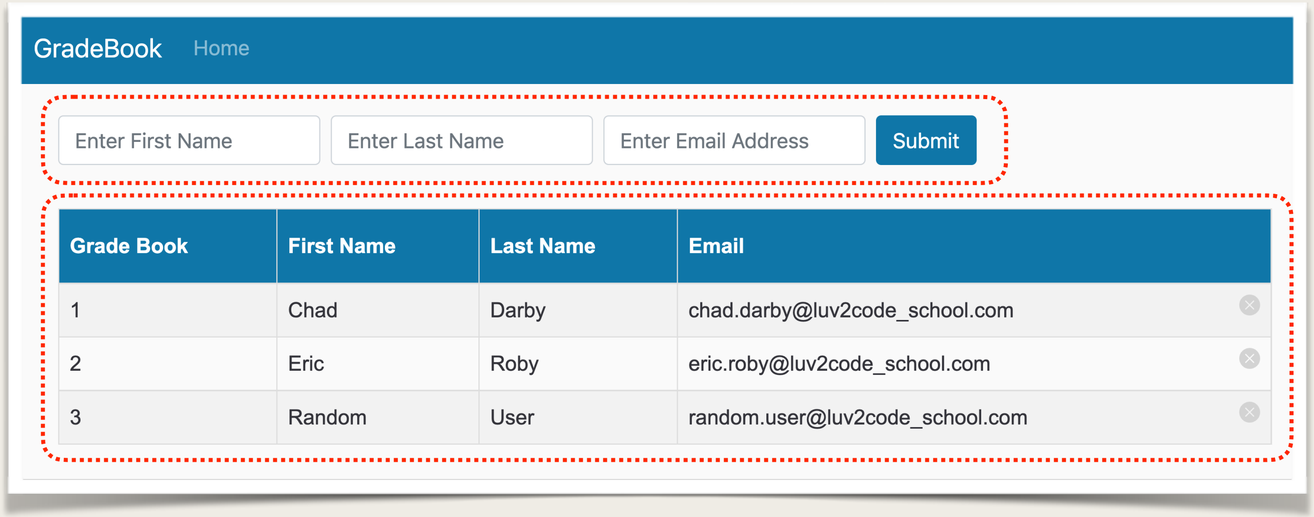
Check Point

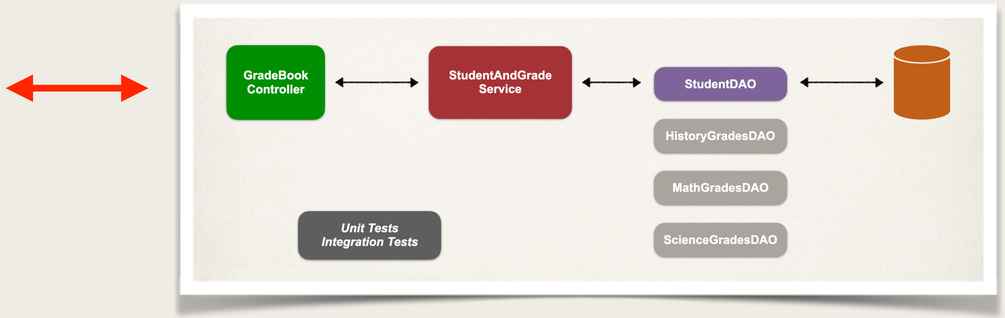
- At the moment, we have developed tests for
  - Creating a student
  - Getting a list of students



Work To Do

- The current UI has hard coded HTML ... doesn't really do anything
- To Do
  - Update index.html to have form submit data to GradeBookController
  - Update index.html to display list of students using a for loop over the "students" model attribute

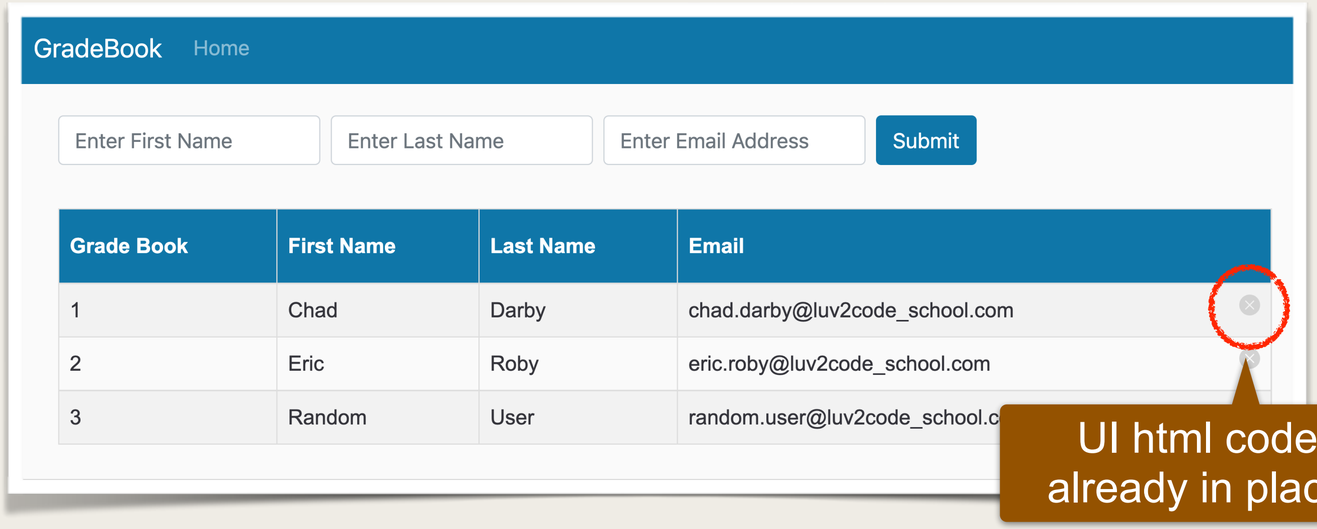


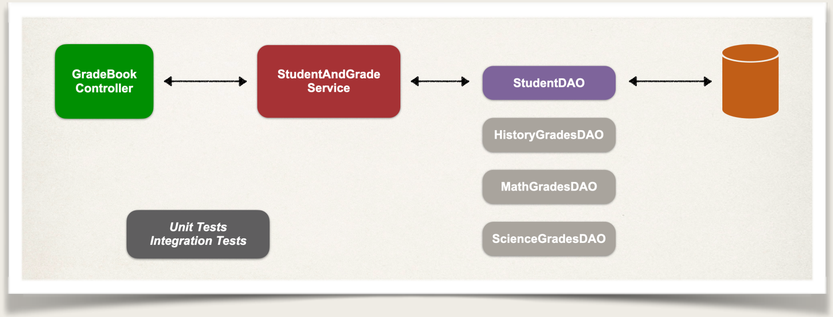


Delete Student

To Do: Delete Student

- Apply TDD
  - Create a failing test
  - Add code to GradeBookController to delete student ... make the test pass
  - Add code to GradeBookController to check for error page ... make the test pass

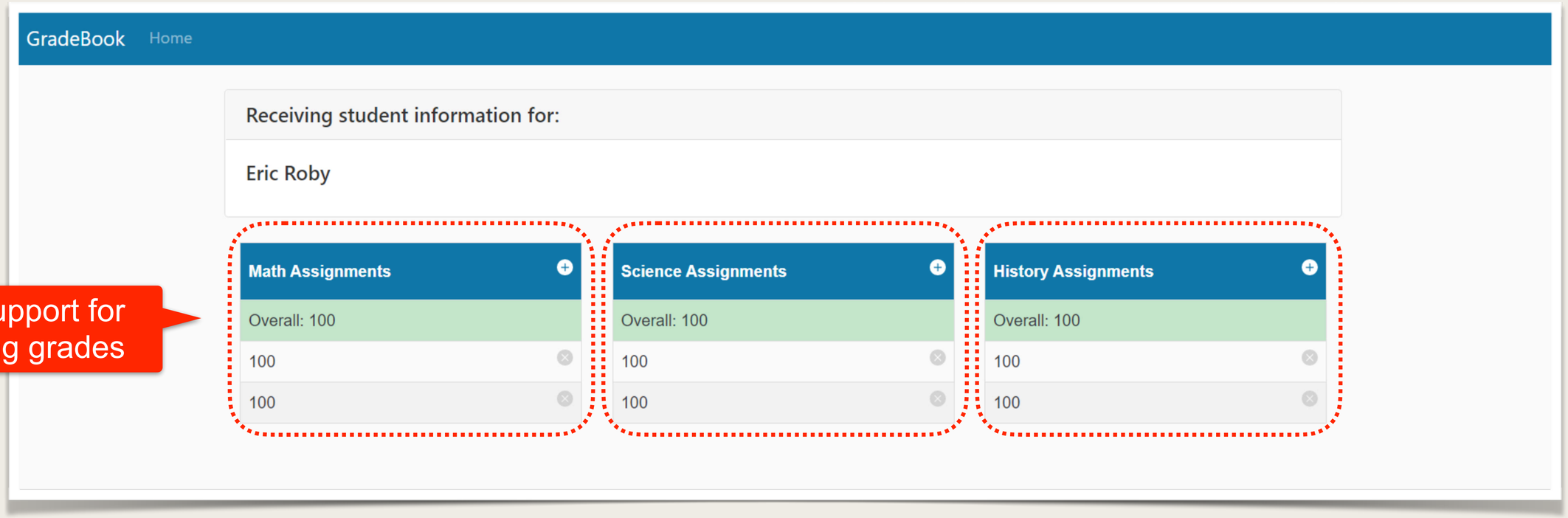




Create Grades

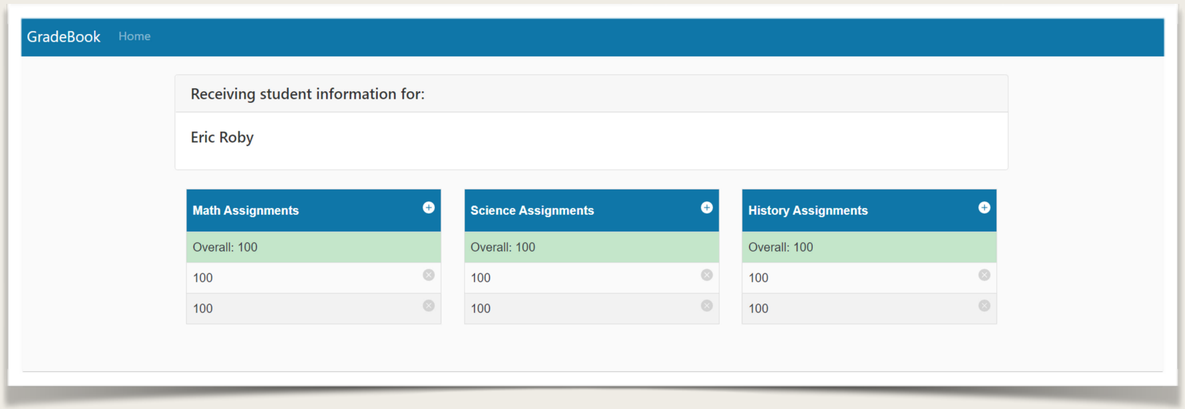
To Do: Grades

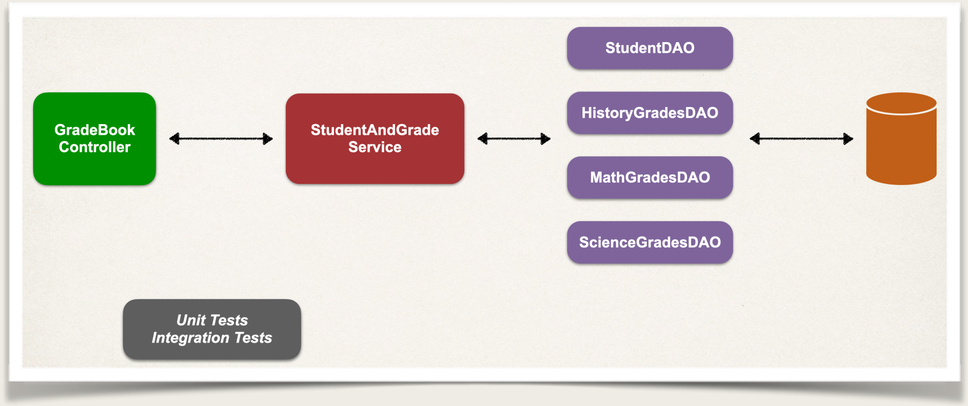
- Currently the app does not keep track of grades for a given student
- At the moment, the UI is hard coded



To Do: Grades

- Apply TDD to add this new functionality
- Update StudentAndGradeService and DAOs to track grades



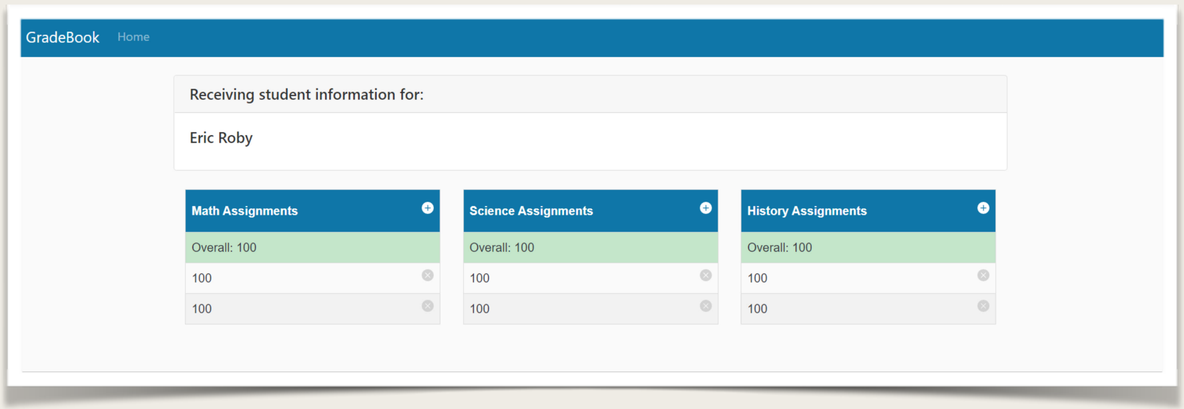


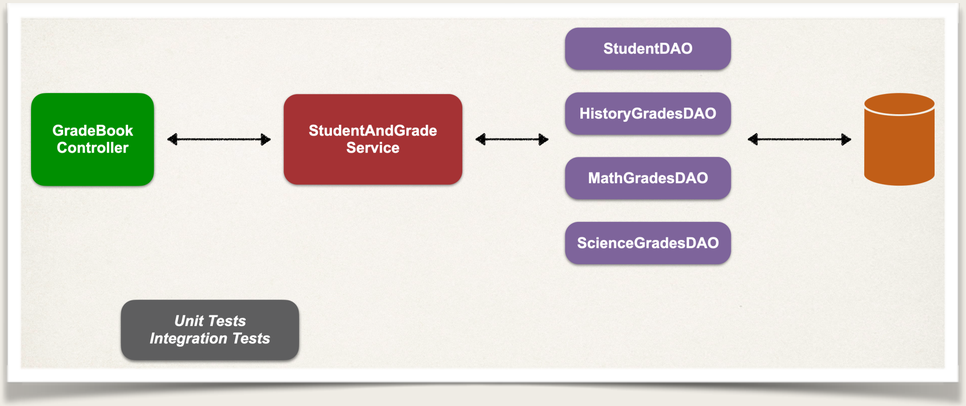
luv2code

Delete Grades

To Do: Delete Grades

- Currently the app does not delete grades
- Apply TDD to implement this new functionality
- Focus on the backend for now ... we'll come back to UI later



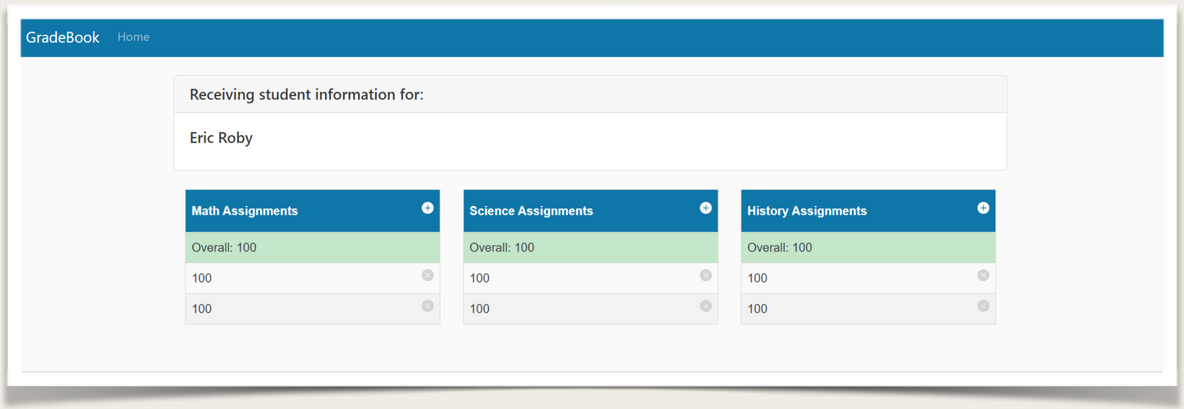


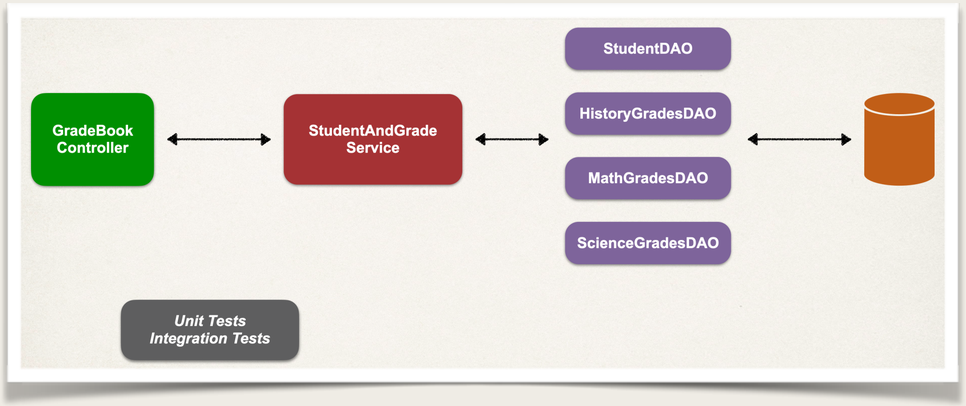
luv2code

Student Information

To Do: Student Information

- Currently the app does not have a method to retrieve student information
  - Student name, email, grades etc
- Apply TDD to implement this new functionality





luv2code

Set Up SQL Scripts in properties file

SQL for Sample Data

- Currently the SQL for sample data is hard-coded in our tests
- We would like to move the SQL to our properties file

@BeforeEach and @AfterEach

StudentAndGradeServiceTest.java

```
import org.springframework.jdbc.core.JdbcTemplate;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
...

@TestPropertySource("/application.properties")
@SpringBootTest
public class StudentAndGradeServiceTest {

   @Autowired
   private JdbcTemplate jdbc;

   @BeforeEach
   public void setupDatabase() {
       jdbc.execute("insert into student(id, firstname, lastname, email_address) " +
               "values (1, 'Eric', 'Roby', 'eric.roby@luv2code_school.com')");
       ...
   }


   @AfterEach
   public void setupAfterTransaction() {
       jdbc.execute("DELETE FROM student");
       ...
   }

}
```

This is what we currently have

Insert sample data

SQL is hard coded

Delete sample data

Development Process

1. Add SQL to application.properties
2. Inject SQL into test using @Value
3. Refactor @BeforeEach and @AfterEach

Step 1: Add SQL to application.properties

application.properties

```
...

sql.script.create.student=insert into student(id,firstname,lastname,email_address) \
  values (1,'Eric', 'Roby', 'eric.roby@luv2code_school.com')

sql.script.delete.student=DELETE FROM student

...
```

Step 2: Inject SQL into test using @Value

StudentAndGradeServiceTest.java

```
...

@TestPropertySource("/application.properties")
@SpringBootTest
public class StudentAndGradeServiceTest {

   @Autowired
   private JdbcTemplate jdbc;

   @Value("${sql.script.create.student}")
   private String sqlAddStudent;

   @Value("${sql.script.delete.student}")
   private String sqlDeleteStudent;

   ...
}
```

application.properties

```
...

sql.script.create.student=insert into student(id,firstname,lastname,email_address) \
  values (1,'Eric', 'Roby', 'eric.roby@luv2code_school.com')

sql.script.delete.student=DELETE FROM student

...
```

Step 3: Refactor @BeforeEach and @AfterEach

StudentAndGradeServiceTest.java

```
...

@TestPropertySource("/application.properties")
@SpringBootTest
public class StudentAndGradeServiceTest {

   @Autowired
   private JdbcTemplate jdbc;

   @Value("${sql.script.create.student}")
   private String sqlAddStudent;

   @Value("${sql.script.delete.student}")
   private String sqlDeleteStudent;

   @BeforeEach
   public void setupDatabase() {
       jdbc.execute(sqlAddStudent);
   }


   @AfterEach
   public void setupAfterTransaction() {
       jdbc.execute(sqlDeleteStudent);
   }

}
```

Refactored code

Refactored code