NOVA SCHOOL OF SCIENCE & TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Internet Applications Design and Implementation

Week 1 - History of Web Apps. Inversion Of Control / Dependency Injection. Spring.

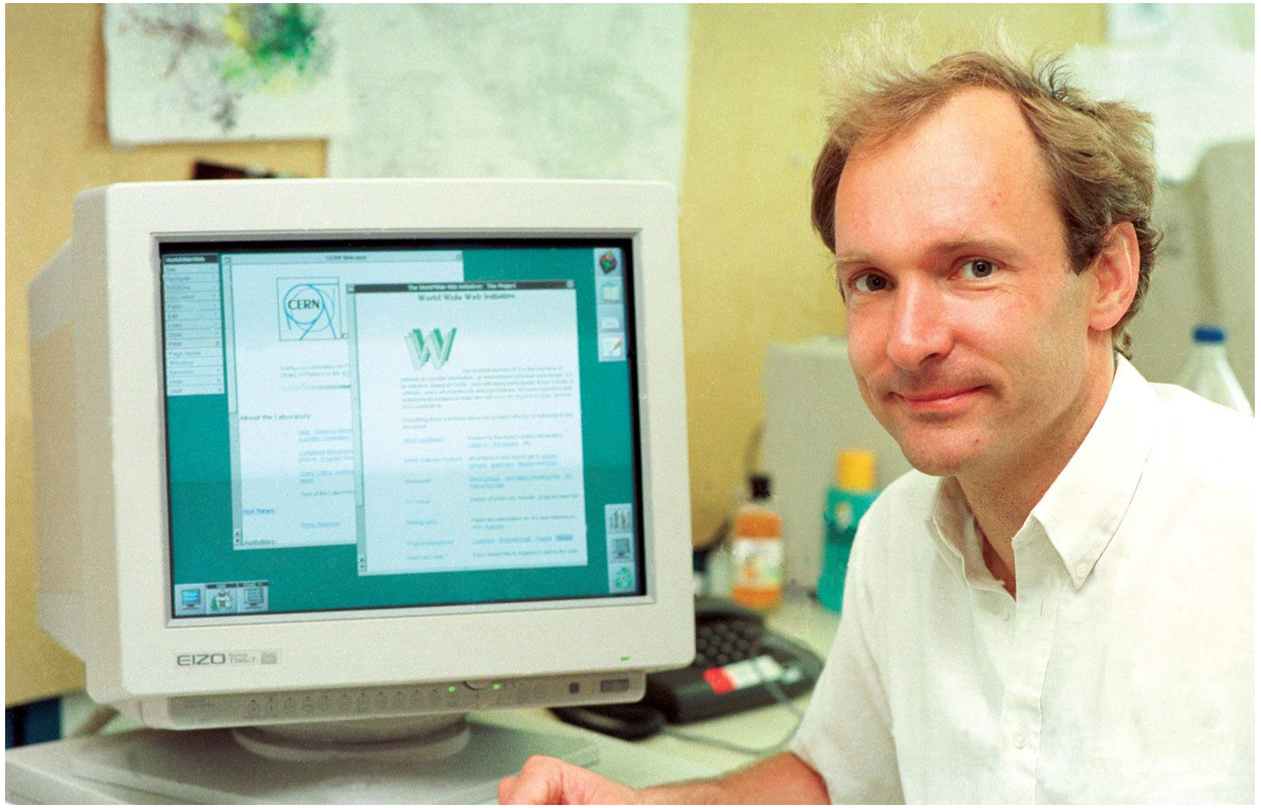
History of Web Apps

heavily based on https://www.youtube.com/watch?v=a_1cV7hg5G8

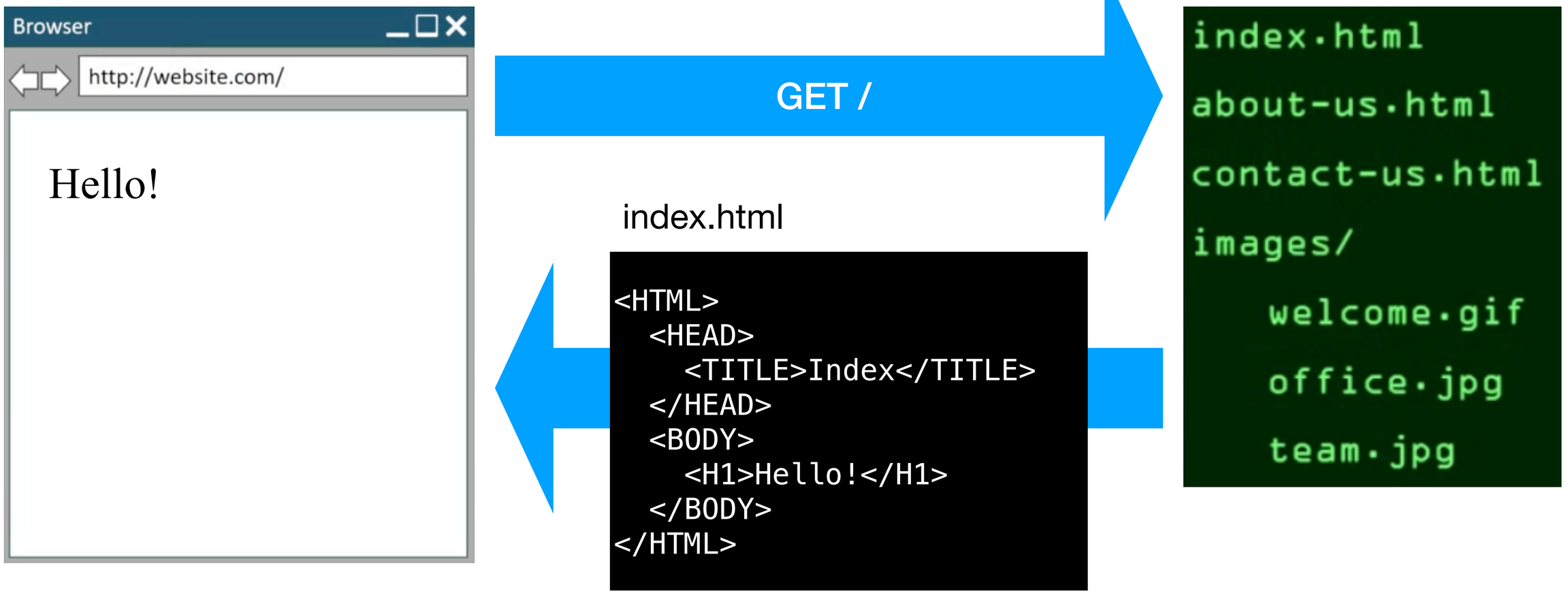
(go watch it)

History of Web Apps

1990 - The beginning



History of Web Apps
1991 - Static HTML



History of Web Apps

1994 - Server side rendering (SSR)

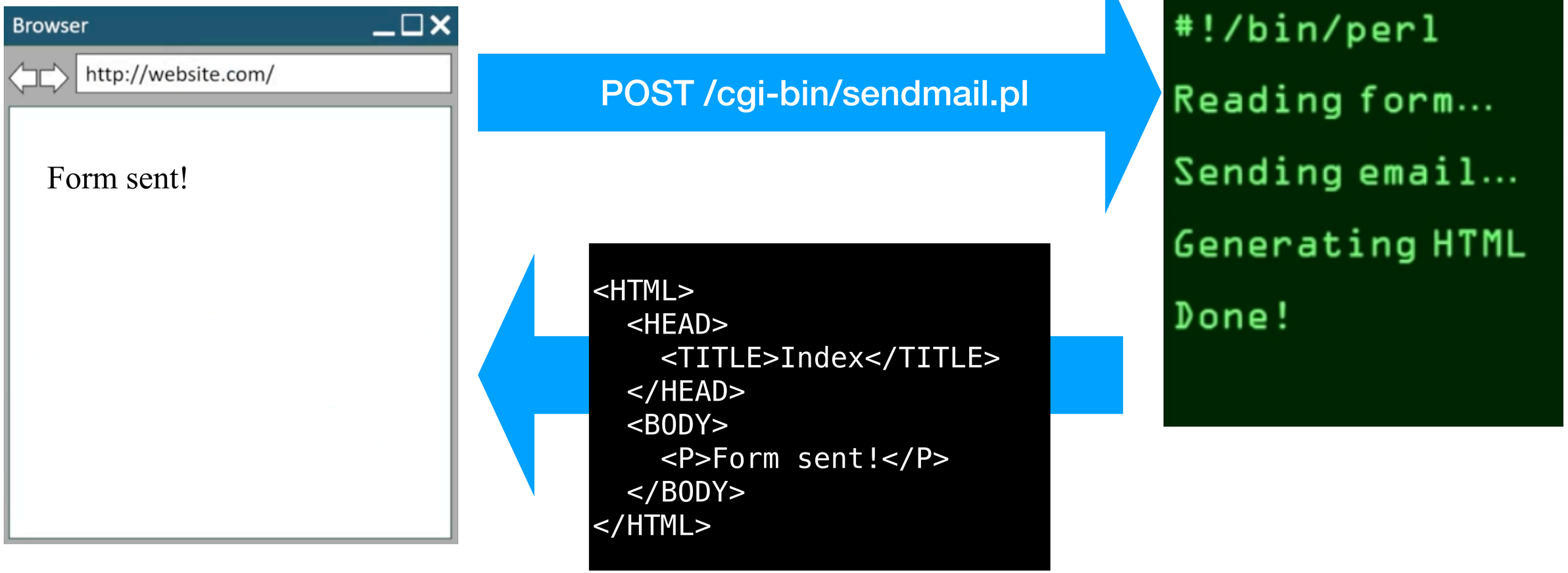
POST /cgi-bin/sendmail.pl

Web Server

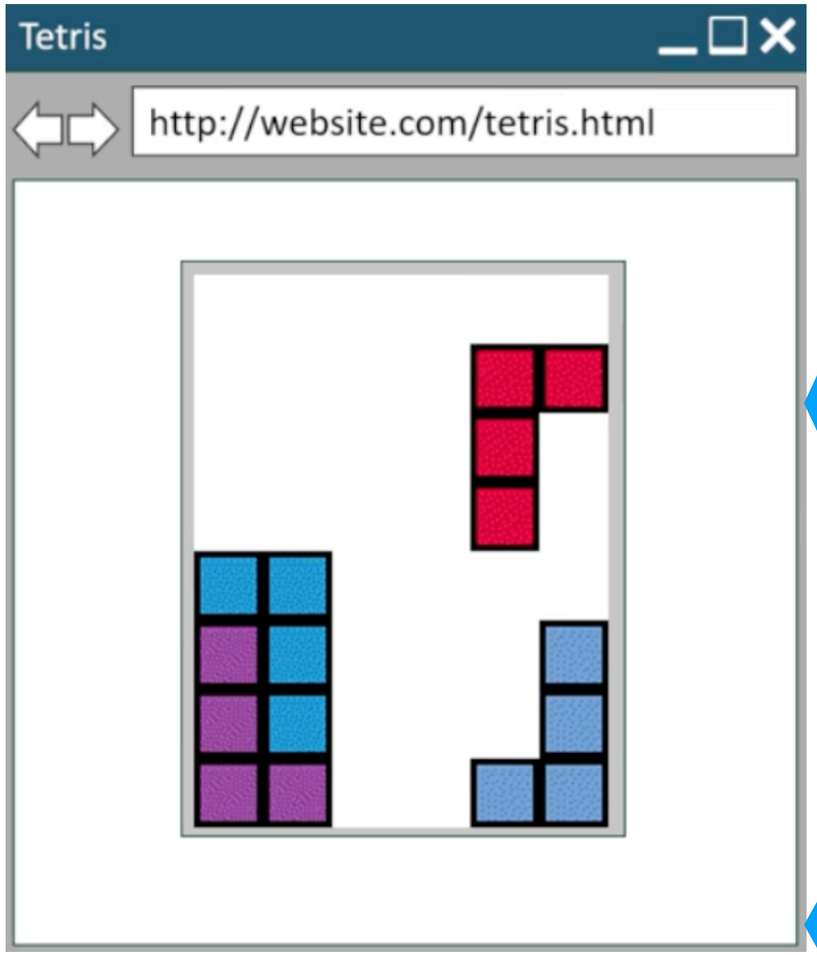
```
#!/bin/perl
Reading form...
Sending email...
Generating HTML
Done!
```

History of Web Apps

1994 - Server side rendering (SSR)



History of Web Apps
1995 - 2012 - Java applets



GET /tetris.html

```
<HTML>
  <HEAD>
    <TITLE>Index</TITLE>
  </HEAD>
  <BODY>
    <APPLET CODE="java/
tetris.class"></APPLET>
  </BODY>
</HTML>
```

GET /java/tetris.class

tetris.class

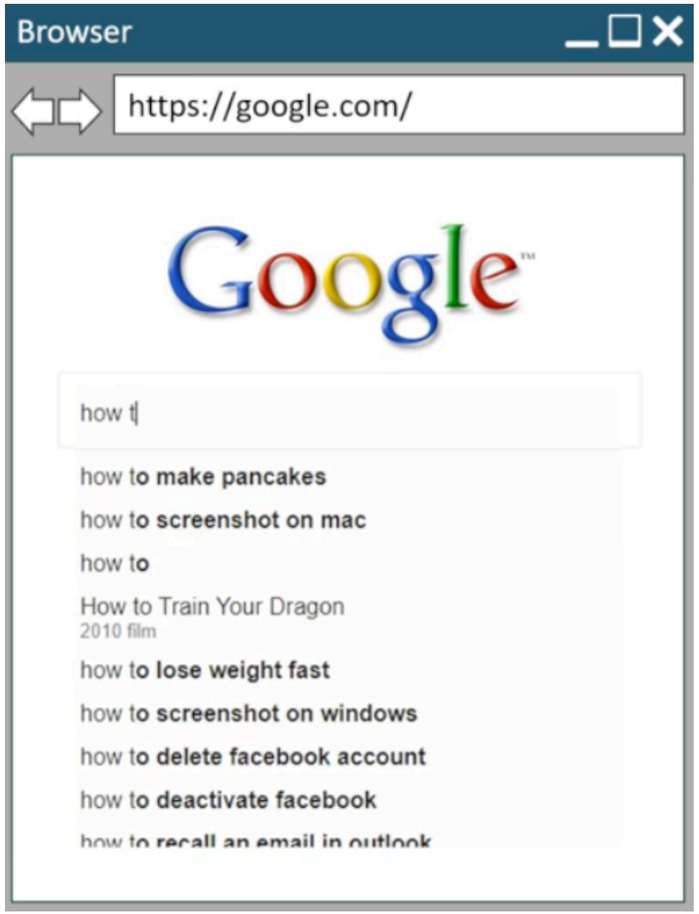
Web Server

```
tetris.html
java/
  tetris.class
```

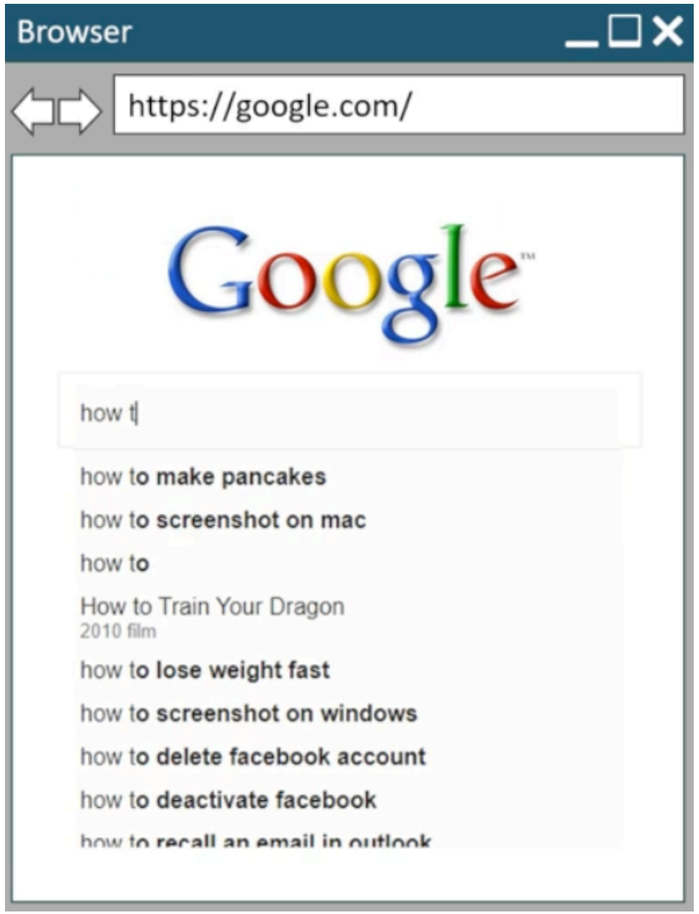
History of Web Apps
1995 - 1999 - limited Javascript

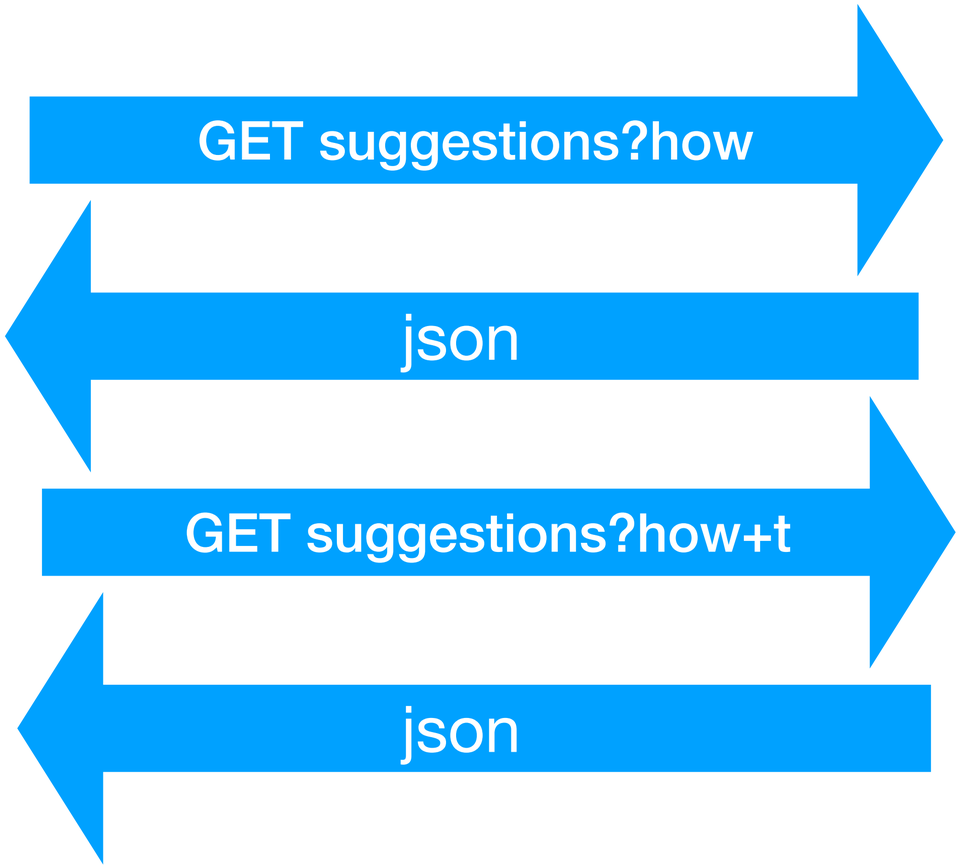

```
<HTML>
  <HEAD>
    <TITLE>Index</TITLE>
  </HEAD>
  <BODY>
    <FORM>
       <LABEL>Name:</LABEL><BR>
       <INPUT TYPE="TEXT" NAME="name"><BR>
       …
       <INPUT TYPE="SUBMIT" VALUE="SUBMIT"
          ONCLICK="return(confirm('Are you sure?'));"
    </FORM>
  </BODY>
```

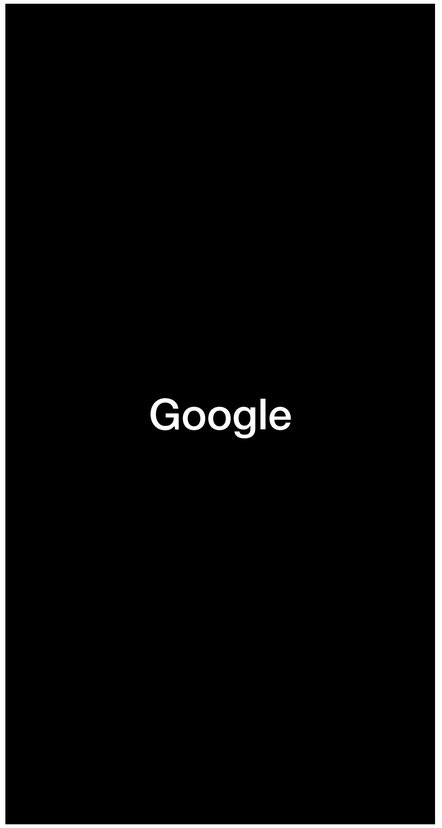
History of Web Apps
2005 - AJAX/Fetch



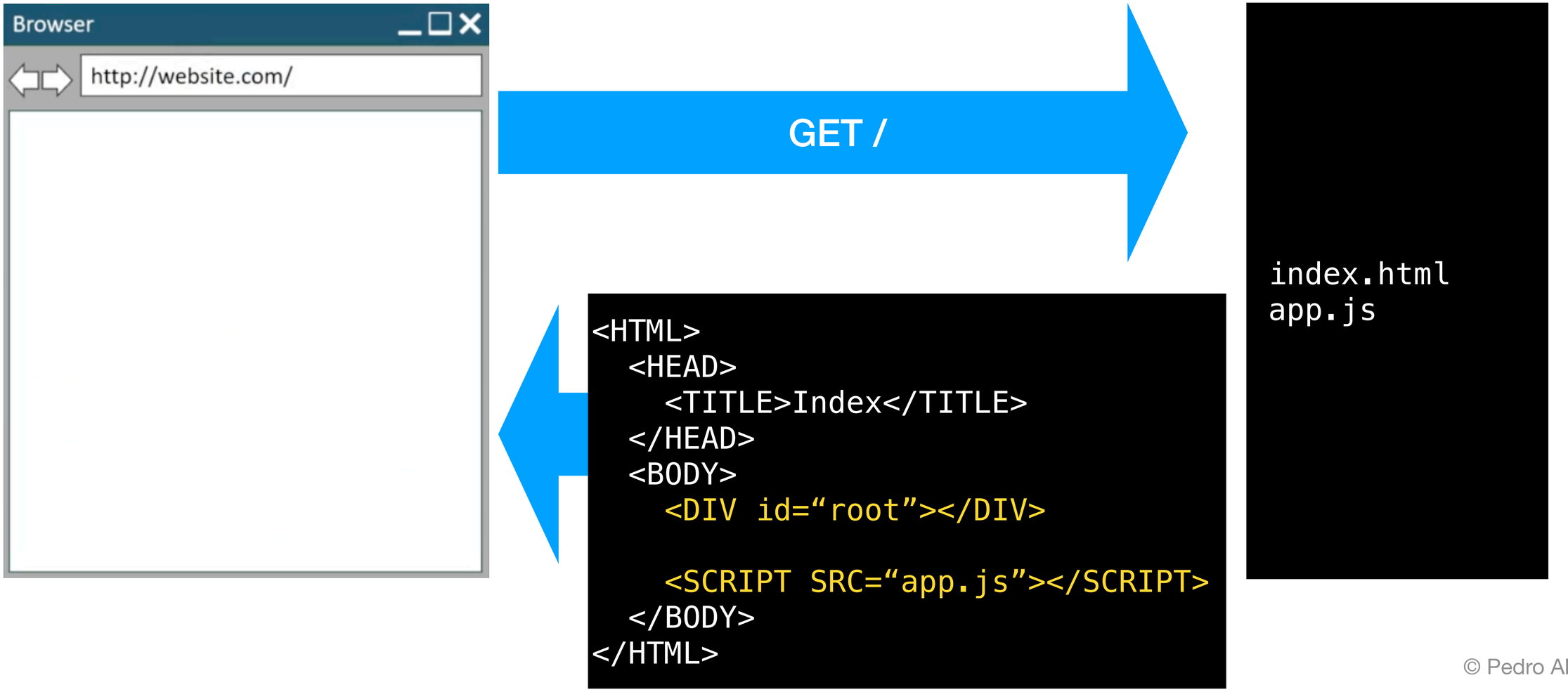
History of Web Apps
2005 - AJAX/Fetch



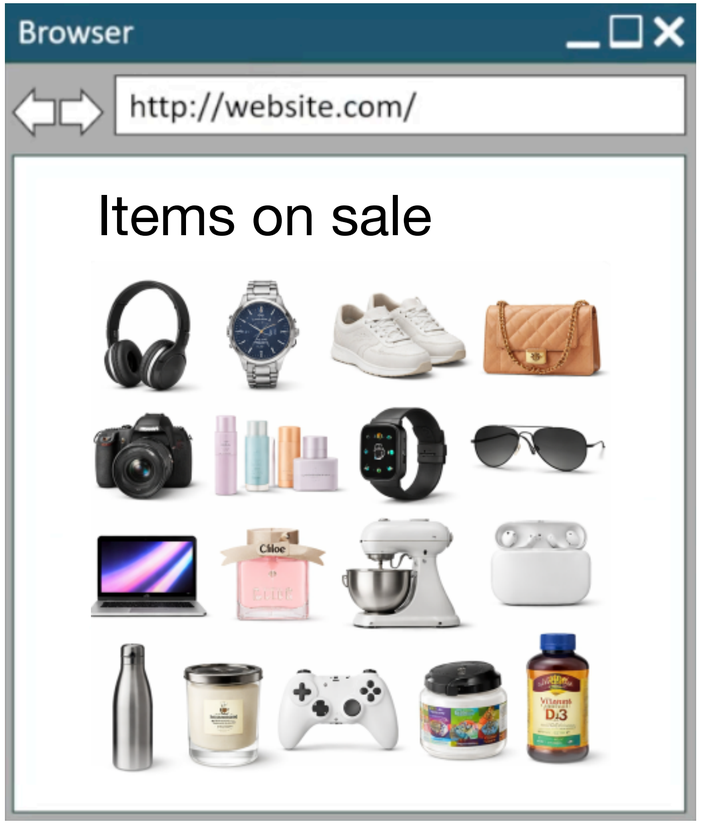




History of Web Apps
2010 - Client side rendering

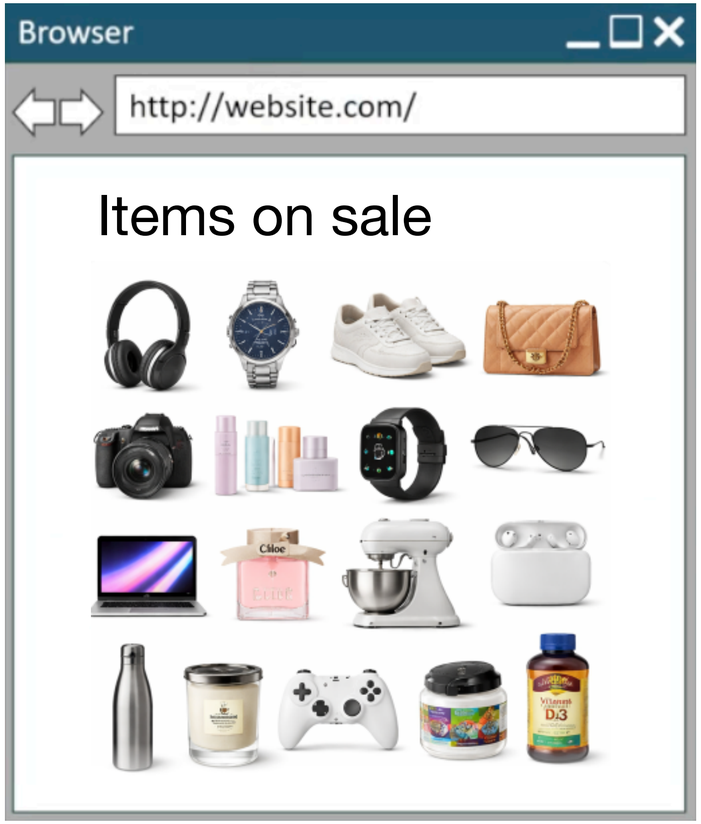


History of Web Apps
2010 - Client side rendering

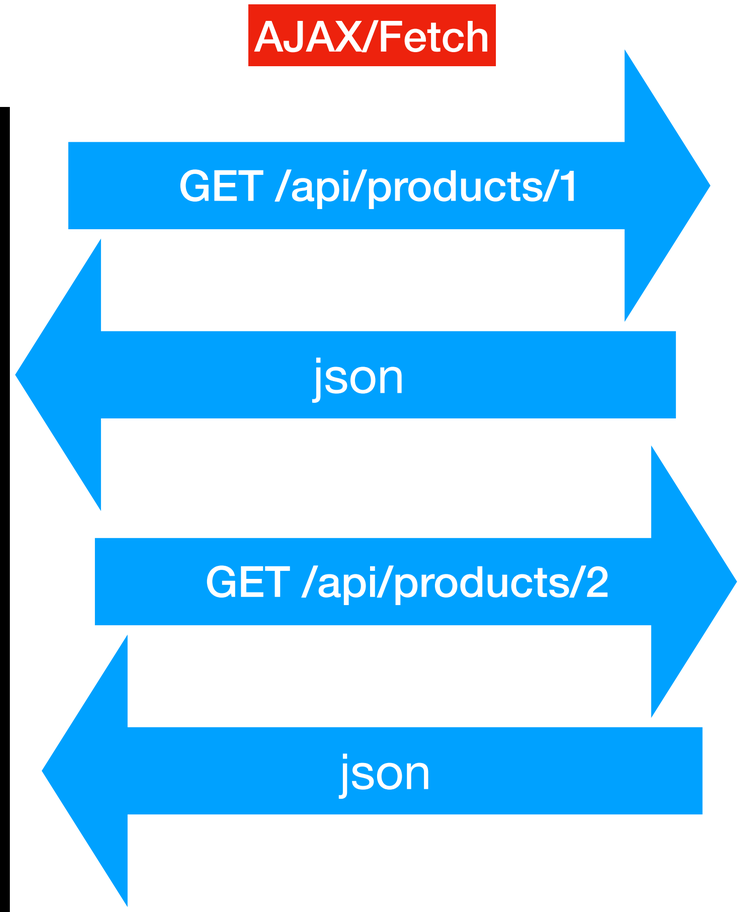



```
<HTML>
  <HEAD>
    <TITLE>Index</TITLE>
  </HEAD>
  <BODY>
    <DIV id="root">
      <DIV id="product1">…
      <DIV id="product2">…
      …
    </DIV>

    <SCRIPT SRC="app.js"></SCRIPT>
  </BODY>
</HTML>
```

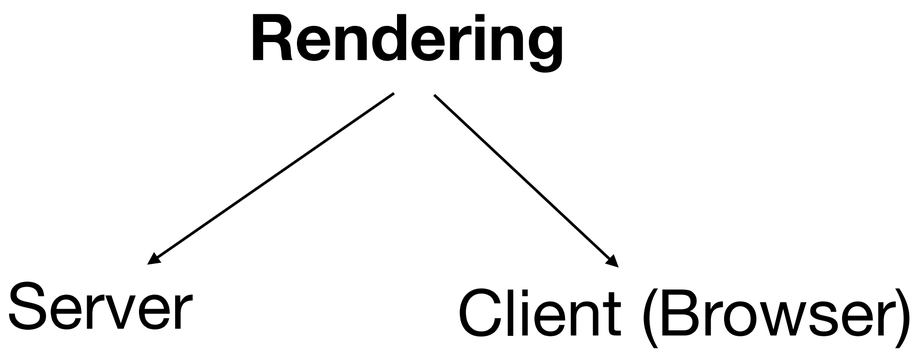
History of Web Apps
2010 - Client side rendering

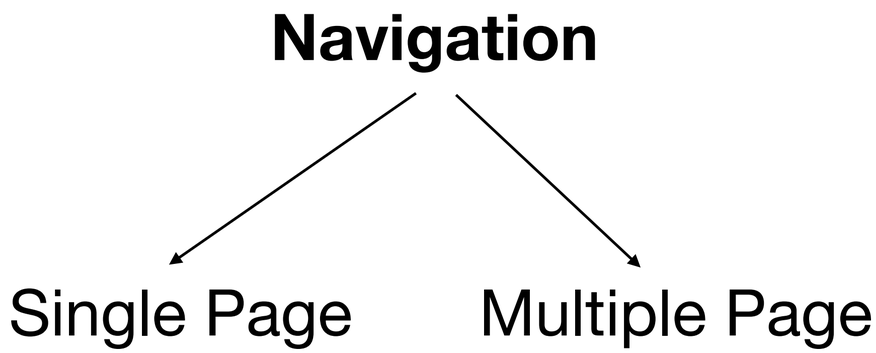



```
<HTML>
  <HEAD>
    <TITLE>Index</TITLE>
  </HEAD>
  <BODY>
    <DIV id="root">
      <DIV id="product1">…
      <DIV id="product2">…
      …
    </DIV>

    <SCRIPT SRC="app.js"></SCRIPT>
  </BODY>
</HTML>
```



Client side rendering ≠ Single Page Apps

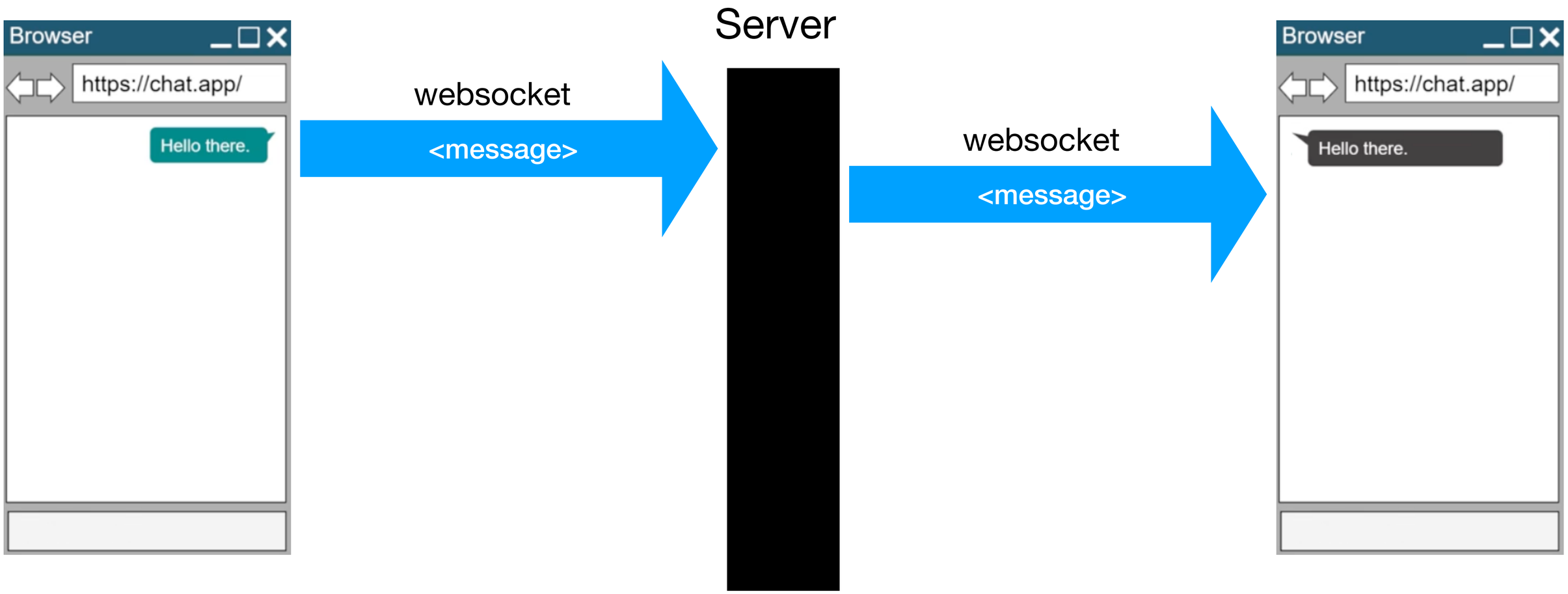




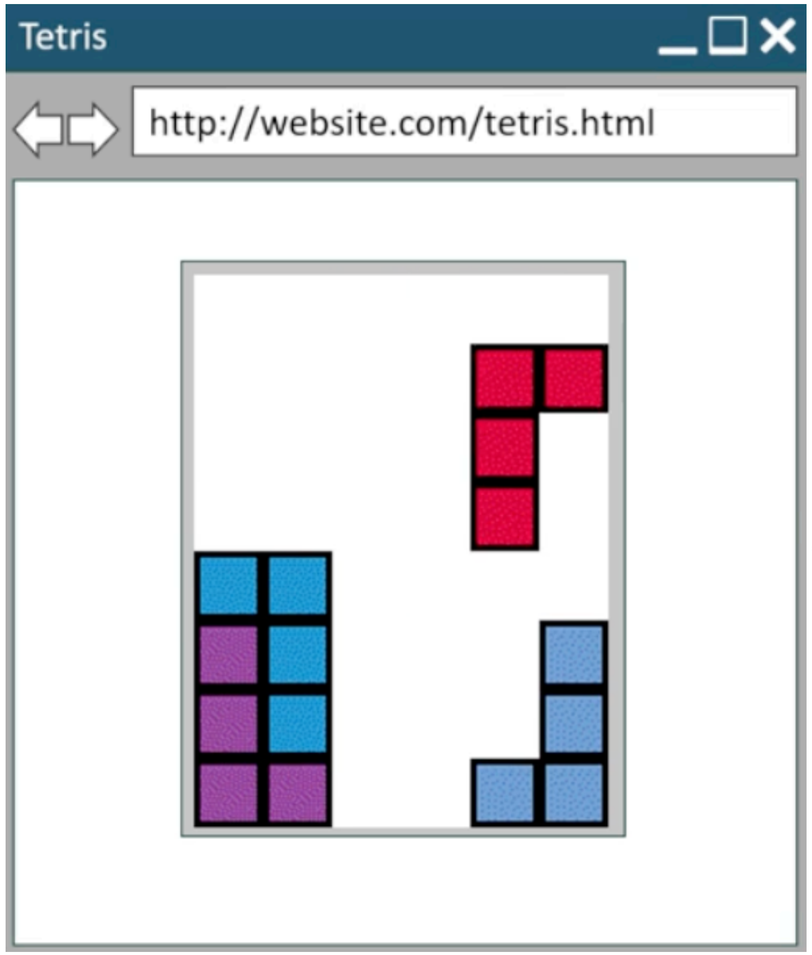
History of Web Apps
2010 until now - History API + SPA Routing


```html
<!doctype html>
<h1 id="h"></h1>
<a href="/1" onclick="go(event,'/1')">Page 1</a>
<a href="/2" onclick="go(event,'/2')">Page 2</a>

<script>
    const h = document.getElementById('h');

    function render() {
        h.textContent = location.pathname === '/2' ? 'Page 2' : 'Page 1';
    }

    function go(e, path) {
        e.preventDefault();
        history.pushState(null, '', path);
        render();
    }

    addEventListener('popstate', render); // back/forward
    render(); // initial
</script>
```

History of Web Apps
2012 - WebSockets



History of Web Apps
2017 - WebAssembly (Wasm)

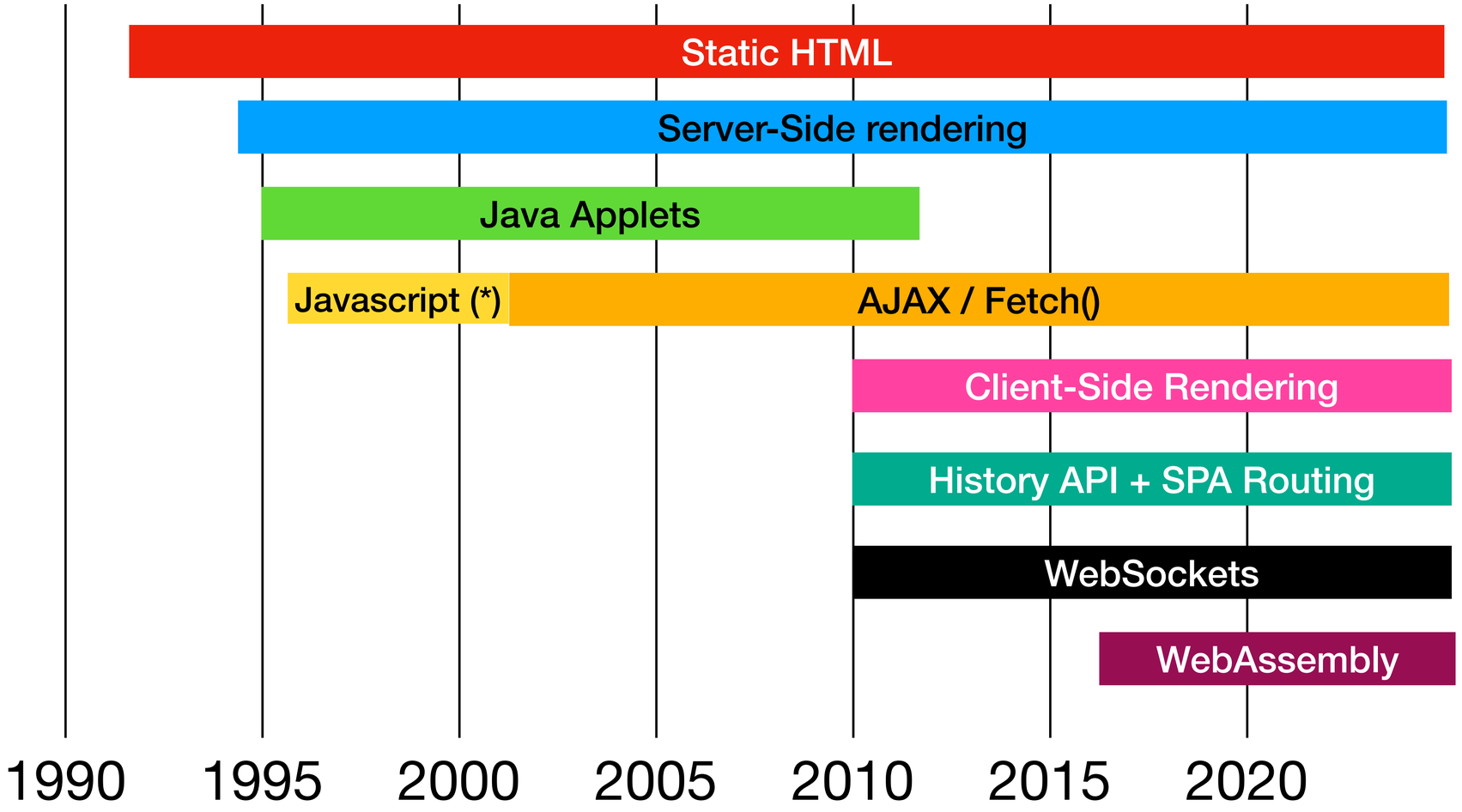


GET /tetris.html

```
<script>
  WebAssembly.
   instantiateStreaming(
     fetch(“tetris.wasm”),
…
</script>
```

Web Server

History of Web Apps
Summary



(*) Limited Javascript

Group Exercise

Approach	Rendering (server, browser, app, N/A)	Page reloads? (always, never, N/A)	Requests style (sync, async, duplex, N/A)	Benefits	Problems
Static HTML					
SSR					
Java Applets					
AJAX/Fetch					
CSR					
SPA History API					
WebSockets					
WebAssembly					

Disclaimer: We are mixing different dimensions (rendering, navigation, network, …) which shouldn't be directly compared… but it's still a fun exercise

Possible solution

Approach	Rendering (server, browser, app, N/A)	Page reloads? (always, never, N/A)	Requests style (sync, async, duplex, N/A)	Benefits	Problems
Static HTML	server (prebuilt files)	always	sync	instant, cacheable	no interactivity
SSR	server	always (1)	sync	dynamic	server processing cost
Java Applets	app	never (inside applet)	N/A	rich UI	not available today
AJAX/Fetch	N/A	never	async	partial updates	possible call explosion
CSR	browser	N/A	N/A	app-like feel	slow first load, SEO challenges
SPA History API	browser	never	N/A	instant navigation	complex URL/state synchronization
WebSockets	N/A	N/A	persistent duplex	real-time updates	server load
WebAssembly	browser	never (inside app)	N/A	near-native speed	limited access, interop overhead

(1) In most cases! In some cases, a page can be partially rendered on the server and filled in on the client (hydration)

Application Framework

IKEA Process:

- Choose furniture in the store
- Bring home only the pieces of that particular furniture
- Assemble (hopefully!) the pieces

Application Framework

Application Framework

- Equivalent to IKEA warehouse
- All the possible pieces for every kind of furniture

The challenge is picking up the right pieces for a given furniture

Spring

Why it exists

- Created in 2004, by Rod Johnson
- Lightweight alternative to Java Platform Enterprise Edition
- Simplifies enterprise Java development

Spring

Core ideas

- Dependency Injection / Inversion of Control (DI/IoC)
- Loose coupling & testability
- Reduce boilerplate / infrastructure code
- Focus on business logic

Spring

What it provides

- Bean & dependency management
- Web + REST support
- Database & transaction handling
- Testing support (unit & integration testing)
- Rapid development (especially with Spring Boot)

Dependency Injection

Motivating Problem

1. Get recent posts from the database
2. Rank those posts with a trending algorithm
3. Notify users of those posts

Dependency Injection

Motivating Problem

```
class TrendingService {

    private val postRepository: SqlPostRepository
    private val algorithm: WeightedTrendingAlgorithm
    private val notifier: SmtpEmailNotificationService

    init {
        val db = DatabaseConnection("jdbc:prod-url", "prodUser", "prodPass")

        postRepository = SqlPostRepository(db)
        algorithm = WeightedTrendingAlgorithm()
        notifier = SmtpEmailNotificationService(
            host = "smtp.gmail.com",
            username = "prod@social.com",
            password = "password"
        )
    }

    fun notifyTrendingPosts() {
        val recentPosts = postRepository.recentPosts()
        val trending = algorithm.rank(recentPosts)

        trending.forEach { post ->
            notifier.notify(post)
        }
    }
}
```

What's wrong with this solution?

Dependency Injection

Motivating Problem

```
class TrendingService {

    private val postRepository: SqlPostRepository
    private val algorithm: WeightedTrendingAlgorithm
    private val notifier: SmtpEmailNotificationService

    init {
        val db = DatabaseConnection("jdbc:prod-url", "prodUser", "prodPass")

        postRepository = SqlPostRepository(db)
        algorithm = WeightedTrendingAlgorithm()
        notifier = SmtpEmailNotificationService(
            host = "smtp.gmail.com",
            username = "prod@social.com",
            password = "password"
        )
    }

    fun notifyTrendingPosts() {
        val recentPosts = postRepository.recentPosts()
        val trending = algorithm.rank(recentPosts)

        trending.forEach { post ->
            notifier.notify(post)
        }
    }
}
```

Problem 1: How do I test this?

- Test with a real DB??
- Send real emails??

Dependency Injection

Motivating Problem

```
class TrendingService {

    private val postRepository: SqlPostRepository
    private val algorithm: WeightedTrendingAlgorithm
    private val notifier: SmtpEmailNotificationService

    init {
        val db = DatabaseConnection("jdbc:prod-url", "prodUser", "prodPass")

        postRepository = SqlPostRepository(db)
        algorithm = WeightedTrendingAlgorithm()
        notifier = SmtpEmailNotificationService(
            host = "smtp.gmail.com",
            username = "prod@social.com",
            password = "password"
        )
    }

    fun notifyTrendingPosts() {
        val recentPosts = postRepository.recentPosts()
        val trending = algorithm.rank(recentPosts)

        trending.forEach { post ->
            notifier.notify(post)
        }
    }
}
```

Problem 2: How do I experiment with different implementations?

- Get the posts from a csv file
- A/B test ranking algorithms (some users affected by ranking A and others by ranking B)
- Send push notifications instead of emails

- ...

Dependency Injection

Motivating Problem

```
class TrendingService {

    private val postRepository: SqlPostRepository
    private val algorithm: WeightedTrendingAlgorithm
    private val notifier: SmtpEmailNotificationService

    init {
        val db = DatabaseConnection("jdbc:prod-url", "prodUser", "prodPass")

        postRepository = SqlPostRepository(db)
        algorithm = WeightedTrendingAlgorithm()
        notifier = SmtpEmailNotificationService(
            host = "smtp.gmail.com",
            username = "prod@social.com",
            password = "password"
        )
    }

    fun notifyTrendingPosts() {
        val recentPosts = postRepository.recentPosts()
        val trending = algorithm.rank(recentPosts)

        trending.forEach { post ->
            notifier.notify(post)
        }
    }
}
```

This code is tightly coupled!

Dependency Injection

Motivating Problem

Concrete class coupling - use interfaces instead

```
class TrendingService {

    private val postRepository: SqlPostRepository
    private val algorithm: WeightedTrendingAlgorithm
    private val notifier: SmtpEmailNotificationService

    init {
        val db = DatabaseConnection("jdbc:prod-url",
                            “prodUser", "prodPass")

        postRepository = SqlPostRepository(db)
        algorithm = WeightedTrendingAlgorithm()
        notifier = SmtpEmailNotificationService(
            host = "smtp.gmail.com",
            username = "prod@social.com",
            password = "password"
        )
    }

    (…)
}
```

```
class TrendingService {

    private val postRepository: PostRepository
    private val algorithm: TrendingAlgorithm
    private val notifier: NotificationService

    init {
        val db = DatabaseConnection("jdbc:prod-url",
                            “prodUser", "prodPass")

        postRepository = SqlPostRepository(db)
        algorithm = WeightedTrendingAlgorithm()
        notifier = SmtpEmailNotificationService(
            host = "smtp.gmail.com",
            username = "prod@social.com",
            password = "password"
        )
    }

    (…)
}
```

Dependency Injection

Motivating Problem

Construction coupling - don’t create the objects, receive them

```
class TrendingService {

    private val postRepository: PostRepository
    private val algorithm: TrendingAlgorithm
    private val notifier: NotificationService

    init {
        val db = DatabaseConnection("jdbc:prod-url",
                                    “prodUser", "prodPass")

        postRepository = SqlPostRepository(db)
        algorithm = WeightedTrendingAlgorithm()
        notifier = SmtpEmailNotificationService(
            host = "smtp.gmail.com",
            username = "prod@social.com",
            password = "password"
        )
    }

    fun notifyTrendingPosts() {
        val recent = repo.recentPosts()
        val trending = algo.rank(recent)

        trending.forEach(notifier::notify)
    }
}
```

```
class TrendingService(
    private val postRepository: PostRepository,
    private val algorithm: TrendingAlgorithm,
    private val notifier: NotificationService
) {

    fun notifyTrendingPosts() {
        val recent = postRepository.recentPosts()
        val trending = algorithm.rank(recent)

        trending.forEach(notifier::notify)
    }
}
```

Dependency Injection

Motivating Problem

```
class TrendingService(
    private val postRepository: PostRepository,
    private val algorithm: TrendingAlgorithm,
    private val notifier: NotificationService
) {

    fun notifyTrendingPosts() {
        val recent = postRepository.recentPosts()
        val trending = algorithm.rank(recent)

        trending.forEach(notifier::notify)
    }
}
```

Testable!!

```
@Test
fun `should notify only trending posts`() {
    val fakeRepo = FakePostRepository(samplePosts)
    val simpleAlgo = SimpleTrendingAlgorithm()
    val fakeotifier = FakeNotificationService()

    val service = TrendingService(
        fakeRepo,
        simpleAlgo,
        fakeNotifier
    )

    service.notifyTrendingPosts()

    assertEquals(3, notifier.sentCount)
}
```

Dependency Injection

Motivating Problem

```
class TrendingService(
    private val postRepository: PostRepository,
    private val algorithm: TrendingAlgorithm,
    private val notifier: NotificationService
) {

    fun notifyTrendingPosts() {
        val recent = postRepository.recentPosts()
        val trending = algorithm.rank(recent)

        trending.forEach(notifier::notify)
    }
}
```

Swapable!! (just change the profile in runtime)

```
@Profile("weighted")
@Component
class WeightedTrendingAlgorithm : TrendingAlgorithm {
    override fun rank(posts: List<Post>): List<Post> {
        return posts.sortedByDescending { it.likes * 2 + it.comments }
    }
}

@Profile("recency")
@Component
class RecencyTrendingAlgorithm : TrendingAlgorithm {
    override fun rank(posts: List<Post>): List<Post> {
        return posts.sortedByDescending { it.createdAt } // newest first
    }
}
```

Inversion of Control

Motivating Problem

```
fun main() {

    val db = DatabaseConnection("jdbc:prod-url", "prodUser", "prodPass")

    val repo: PostRepository = SqlPostRepository(db)

    val algo: TrendingAlgorithm = WeightedTrendingAlgorithm()

    val notifier: EmailNotificationService =
        SmtpEmailNotificationService(
            "smtp.gmail.com", "prod@social.com", "password");

    val service = TrendingService(repo, algo, notifier)

    service.notifyTrendingPosts()
}
```

What's wrong with this solution?

Inversion of Control

Motivating Problem

```
fun main() {

    val db = DatabaseConnection("jdbc:prod-url", "prodUser", "prodPass")

    val repo: PostRepository = SqlPostRepository(db)

    val algo: TrendingAlgorithm = WeightedTrendingAlgorithm()

    val notifier: EmailNotificationService =
        SmtpEmailNotificationService(
            "smtp.gmail.com", "prod@social.com", "password");

    val service = TrendingService(repo, algo, notifier)

    service.notifyTrendingPosts()
}
```

Problem 1: Does it scale to dozens of services?

- Explosion of dependencies to manage
- Lots of object creation

Inversion of Control

Motivating Problem

```
fun main() {

    val db = DatabaseConnection("jdbc:prod-url", "prodUser", "prodPass")

    val repo: PostRepository = SqlPostRepository(db)

    val algo: TrendingAlgorithm = WeightedTrendingAlgorithm()

    val notifier: EmailNotificationService =
        SmtpEmailNotificationService(
            "smtp.gmail.com", "prod@social.com", "password");

    val service = TrendingService(repo, algo, notifier)

    service.notifyTrendingPosts()
}
```

Problem 2: Deciding which implementation must be coded?

- A/B testing resulting in if's scattered around the code

Inversion of Control & Dependency Injection

Solution

```
fun main() {

    runApplication<SocialApp>().use { context ->

        val service = context.getBean<TrendingService>()

        service.notifyTrendingPosts()
    }
}
```

Inversion of Control & Dependency Injection

Solution

```
fun main() {

    runApplication<SocialApp>().use { context ->

        val service = context.getBean<TrendingService>()

        service.notifyTrendingPosts()
    }
}
```

A Spring Container takes control of the application:

- Creates the objects (SqlPostRepository, WeightedTrendingAlgorithm, SmtpEmailNotificationService)
- Injects dependencies (injects the above objects into TrendingService)
- Reads configuration files where the database and smtp properties are stored
- Swaps implementations based on profiles, configuration, etc.

Exercise

Find errors in this code

```
class ModerationService(
    private val repo: PostRepository,
    private val notifier: EmailNotificationService
) {

    private val profanityFilter = ProfanityFilter("badwords.txt")
    private val toxicityModel = ToxicityModel("/models/toxicity.bin")

    fun moderate(postId: String) {
        val post = repo.find(postId) ?: return

        val containsProfanity = profanityFilter.containsBadWords(post.text)
        val toxicityScore = toxicityModel.score(post.text)

        val calculator = ThresholdCalculator()

        val shouldReject = calculator.isAboveThreshold(
            score = toxicityScore,
            threshold = 0.85
        ) || containsProfanity

        if (shouldReject) {
            notifier.notifyModerator(post)
            repo.markApproved(postId)
        } else {
            repo.markRejected(postId)
        }
    }
}
```

```
1.  class ModerationService(
2.      private val repo: PostRepository,
3.      private val notifier: EmailNotificationService
4.  ) {
5.
6.      private val profanityFilter = ProfanityFilter("badwords.txt")
7.      private val toxicityModel = ToxicityModel("/models/toxicity.bin")
8.
9.      fun moderate(postId: String) {
10.         val post = repo.find(postId) ?: return
11.
12.         val containsProfanity = profanityFilter.containsBadWords(post.text)
13.         val toxicityScore = toxicityModel.score(post.text)
14.
15.         val calculator = ThresholdCalculator()
16.
17.         val shouldReject = calculator.isAboveThreshold(
18.             score = toxicityScore,
19.             threshold = 0.85
20.         ) || containsProfanity
21.
22.         if (shouldReject) {
23.             notifier.notifyModerator(post)
24.             repo.markApproved(postId)
25.         } else {
26.             repo.markRejected(postId)
27.         }
28.     }
29. }
```

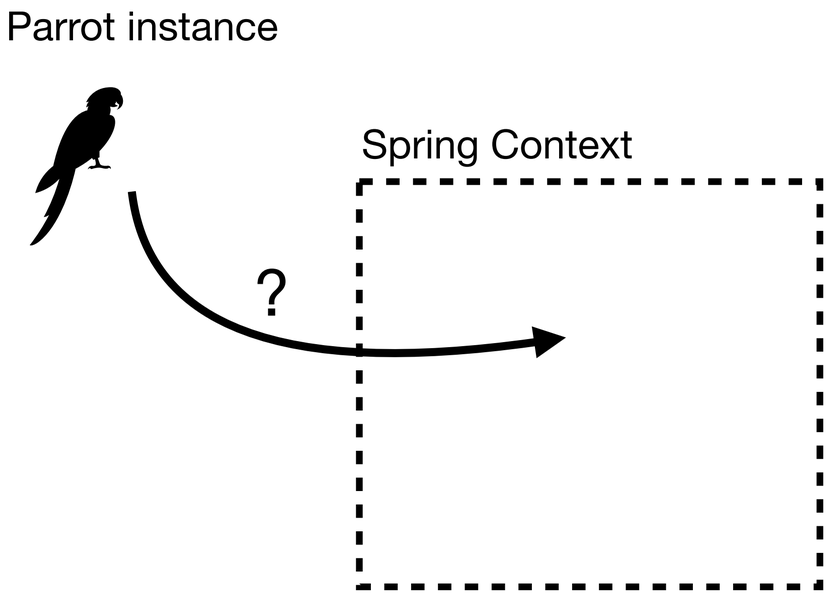
should be NotificationService (interface)

should be injected in the constructor

should be switched

Dependency Injection

Spring Context & Beans



Method #1 - @Bean inside a @Configuration

```
@Configuration
class ProjectConfig {

    @Bean
    fun parrot() = Parrot()
}
```

Notice the method name. The bean will have this name

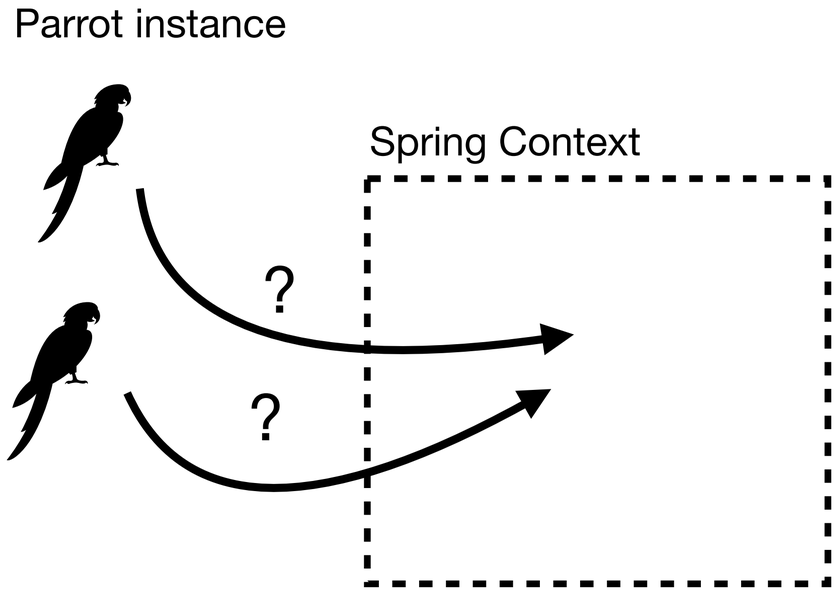
This will create a single instance of Parrot on startup (**singleton**) and put it in the Spring Context

```
fun main(args: Array<String>) {
    val context = runApplication<ExampleApplication>(*args)
    val parrot = context.getBean("parrot")
    val parrot2 = context.getBean("parrot")
}
```

parrot and parrot2 are the same object

Dependency Injection

Spring Context & Beans



Method #1 - @Bean inside a @Configuration

```
@Configuration
class ProjectConfig {

    @Bean
    @Scope("prototype")
    fun parrot() = Parrot()
}
```

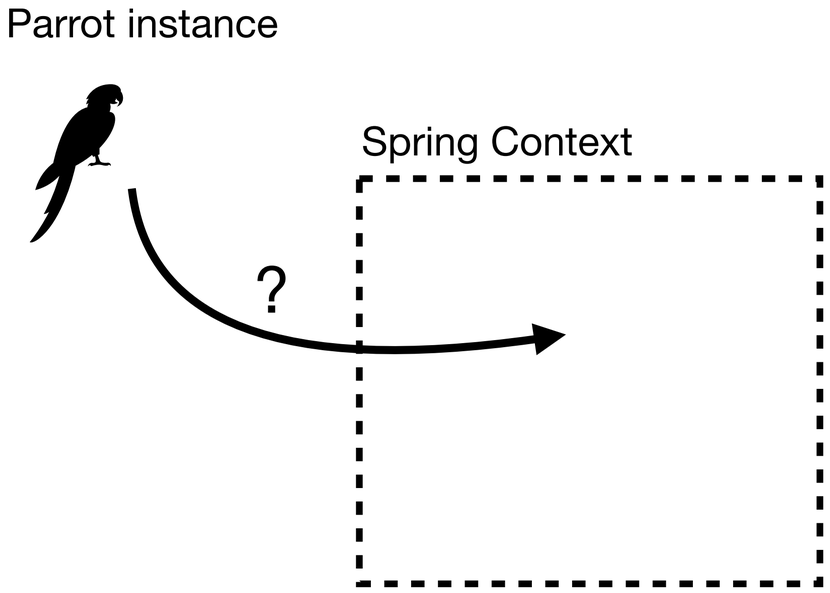
This will create a new instance of Parrot every time someone gets it from Spring Context

```
fun main(args: Array<String>) {
    val context = runApplication<ExampleApplication>(*args)
    val parrot = context.getBean("parrot")
    val parrot2 = context.getBean("parrot")
}
```

parrot and parrot2 are different objects

Dependency Injection

Spring Context & Beans



Method #2 - Stereotype annotations

```
@Component
class Parrot
```

The bean will be named after the class (lowercase)

On startup Spring will scan the project (*) for classes annotated with stereotype annotations (no need for Configuration classes)

This will create a single instance of Parrot on startup (**singleton**) and put it in the Spring Context

You can use @Scope(“prototype") to create several instances

Besides @Component, there is also @Repository, @Service, etc…

Dependency Injection

Spring Context & Beans

@Bean

```
@Configuration
class ProjectConfig {

    @Bean
    fun parrot() = Parrot()
}
```

- Full control over instantiation
- Needs to write a method to create each bean; advisable to use inside a Configuration class

Stereotype annotations

```
@Component
class Parrot
```

- No control, Spring instantiates for you
- Doesn’t add boilerplate code

```
1.  @Component
2.  class User(val id: Long)
3.
4.  @Configuration
5.  class Config {
6.
7.      @Bean
8.      fun createUser1() = User(1)
9.
10.     @Bean
11.     fun user2() = User(2)
12. }
13.
14. fun main(args: Array<String>) {
15.     val context = runApplication<ExampleApplication>(*args)
16.     val user1 = context.getBean("user1") as User
17.     val user2 = context.getBean("user2") as User
18.     val user3 = context.getBean("user2") as User
19.     val user4 = context.getBean("user") as User
20.     println("user1 = ${user1.id}")
21.     println("user2 = ${user2.id}")
22.     println("user3 = ${user3.id}")
23.     println("user4 = ${user4.id}")
24.
25. }
26.
```

Exercise

Compilation errors?

Runtime errors?

What will be the output (assuming no errors)?

```
1.  @Component
2.  class User(val id: Long)
3.
4.  @Configuration
5.  class Config {
6.
7.      @Bean
8.      fun createUser1() = User(1)
9.
10.     @Bean
11.     fun user2() = User(2)
12. }
13.
14. fun main(args: Array<String>) {
15.     val context = runApplication<ExampleApplication>(*args)
16.     val user1 = context.getBean("user1") as User
17.     val user2 = context.getBean("user2") as User
18.     val user3 = context.getBean("user2") as User
19.     val user4 = context.getBean("user") as User
20.     println("user1 = ${user1.id}")
21.     println("user2 = ${user2.id}")
22.     println("user3 = ${user3.id}")
23.     println("user4 = ${user4.id}")
24.
25. }
26.
```

Runtime error: Can't use @Component with args constructor

Runtime error: There is no bean named "user1"

Resolution

No compilation errors

Output assuming no errors:

```
user1 = 1
user2 = 2
user3 = 2
user4 = 0
```

```
@Component
class User(val id: Long = 0L)

@Configuration
class Config {

    @Bean
    fun user1() = User(1)

    @Bean
    fun user2() = User(2)
}

fun main(args: Array<String>) {
    val context = runApplication<ExampleApplication>(*args)
    val user1 = context.getBean("user1") as User
    val user2 = context.getBean("user2") as User
    val user3 = context.getBean("user2") as User
    val user4 = context.getBean("user") as User
    println("user1 = ${user1.id}")
    println("user2 = ${user2.id}")
    println("user3 = ${user3.id}")
    println("user4 = ${user4.id}")
}
```

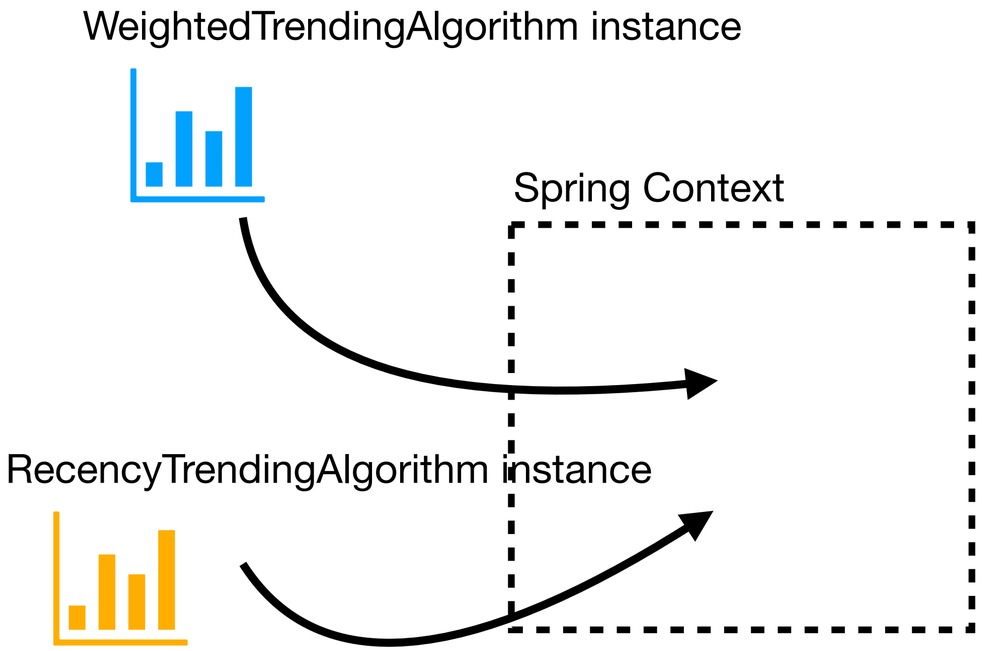
Resolution

Output assuming no errors:

```
user1 = 1
user2 = 2
user3 = 2
user4 = 0
```

Dependency Injection

Spring Context & Beans




```
@Profile("weighted")
@Component
class WeightedTrendingAlgorithm : TrendingAlgorithm {
    override fun rank(posts: List<Post>): List<Post> {
        return posts.sortedByDescending { it.likes * 2 + it.comments }
    }
}

@Profile("recency")
@Component
class RecencyTrendingAlgorithm : TrendingAlgorithm {
    override fun rank(posts: List<Post>): List<Post> {
        return posts.sortedByDescending { it.createdAt }
    }
}
```

Which bean will we get?

```
val algorithm = context.getBean<TrendingAlgorithm>()

algorithm.rank(posts)
```

Dependency Injection

Spring Context & Beans

When there are multiple candidate beans, Spring decides based on:

- @Primary
- @Order
- @Profile
- @ConditionalOnProperty
- …

Dependency Injection

Spring Context & Beans

```
@Profile("weighted")
@Component
class WeightedTrendingAlgorithm : TrendingAlgorithm {
    override fun rank(posts: List<Post>): List<Post> {
        return posts.sortedByDescending { it.likes * 2 + it.comments }
    }
}

@Profile("recency")
@Component
class RecencyTrendingAlgorithm : TrendingAlgorithm {
    override fun rank(posts: List<Post>): List<Post> {
        return posts.sortedByDescending { it.createdAt }
    }
}
```

application.properties

```
spring.profiles.active=recency
```

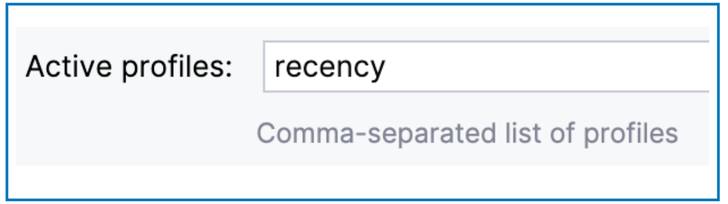
command line

```
java -jar app.jar --spring.profiles.active=recency
```

```
val algorithm = context.getBean<TrendingAlgorithm>()
```

RecencyTrendingAlgorithm

Intellij



Maven

- Standard project structure
- Build lifecycle
- Dependency management
- Plugins
- Automatically recognized by IDEs

Maven

Standard project structure

```
my-project/
├── src/
│   ├── main/
│   │   ├── kotlin/          # Application source code
│   │   └── resources/       # Configuration files, templates
│   └── test/
│       ├── kotlin/          # Test source code
│       └── resources/       # Test configuration files
├── target/                  # Build outputs (generated, not committed)
└── pom.xml                  # Build configuration
```

Maven

Build Lifecycle

- **Clean** — Delete previous build outputs
- **Compile** — Transform source code into executable form
- **Test** — Run automated tests
- **Package** — Bundle the application (JAR, wheel, tarball, etc.)
- **Install** — Place the package in a local repository
- **Deploy** — Publish the package to a remote repository

```
mvn clean compile
```

pom.xml

```xml
<dependencies>
    <!-- compile scope (default) - available at compile and runtime -->
    <dependency>
        <groupId>org.apache.httpcomponents</groupId>
        <artifactId>httpclient</artifactId>
        <version>4.3.6</version>
    </dependency>

    <!-- provided scope - only at compile time, not packaged -->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.30</version>
        <scope>provided</scope>
    </dependency>

    <!-- runtime scope - only at runtime -->
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.6.0</version>
        <scope>runtime</scope>
    </dependency>

    <!-- test scope - only for testing -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.10.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Maven

Dependency management

- Versions
- Scope

Maven

Plugins

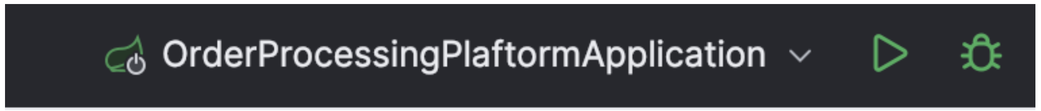
pom.xml

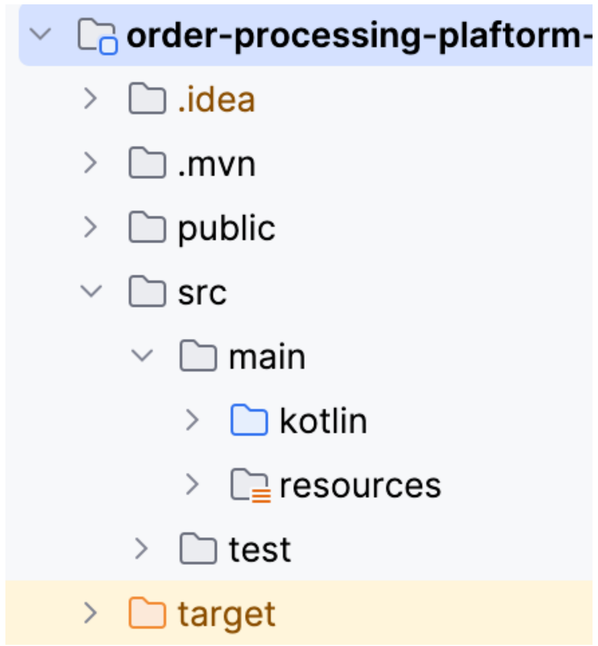
```
<plugins>
    <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
</plugins>
```

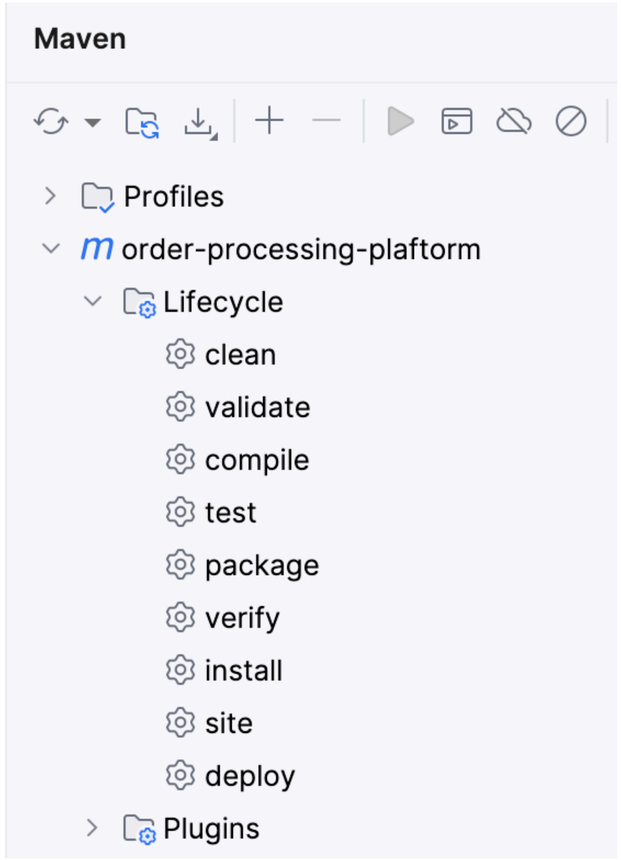
`mvn spring-boot:run` ⟶ compiles, finds the main class, runs the application

Maven

Automatically recognized by IDEs (Intellij)

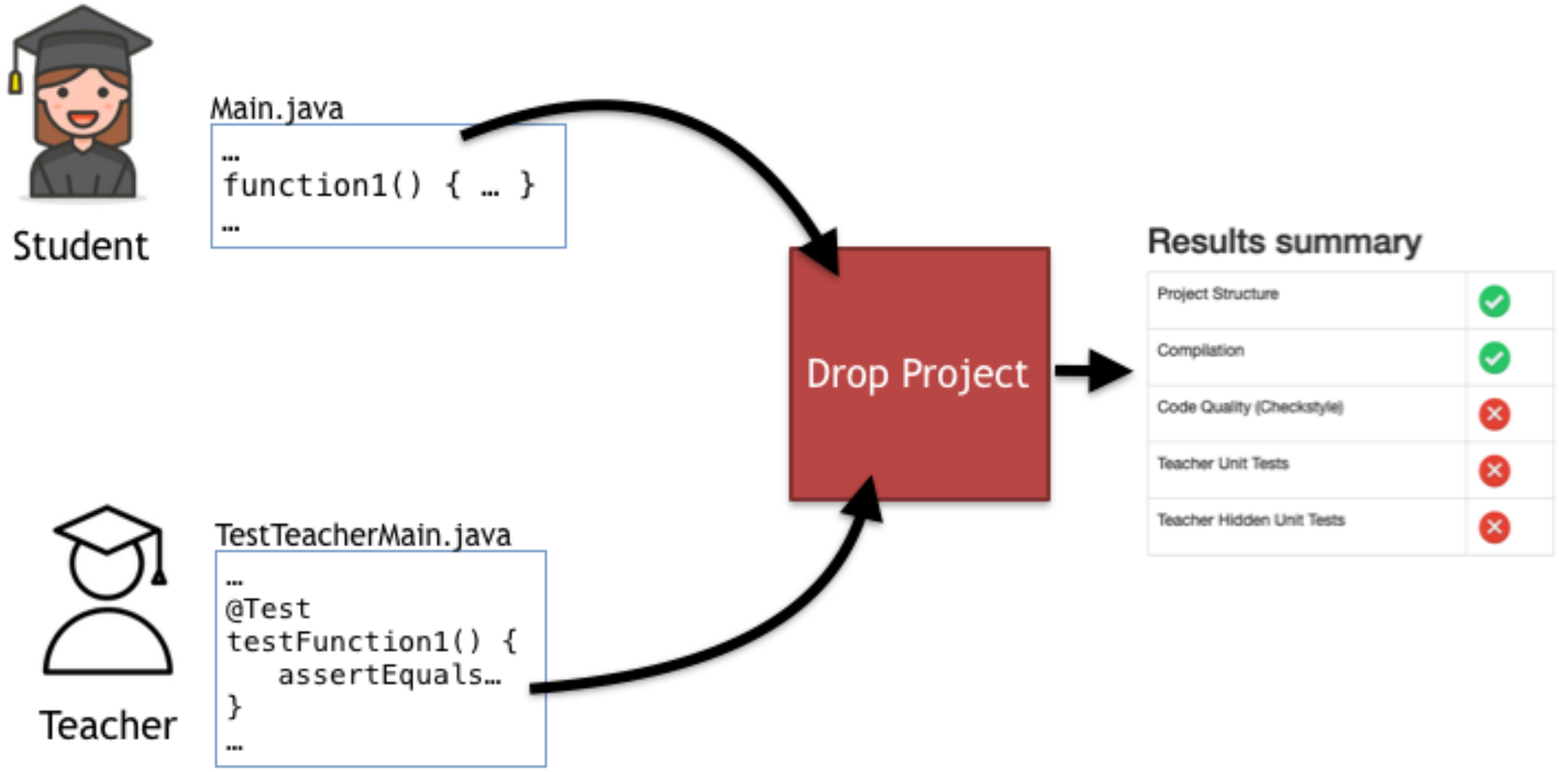






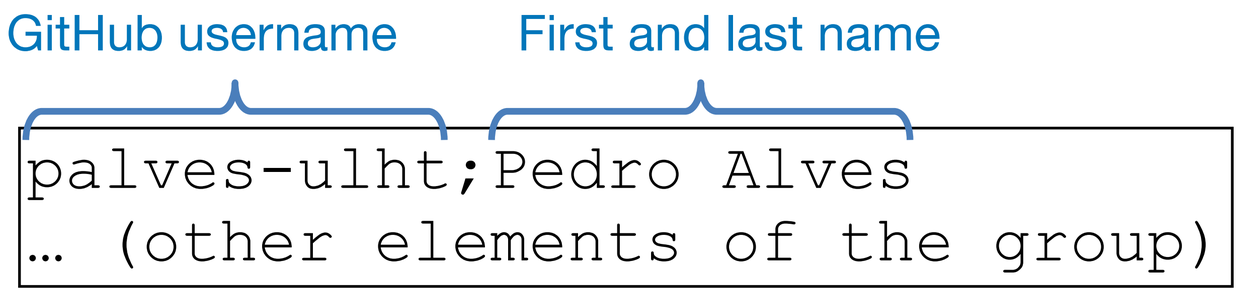
Drop Project

In practical classes you will use Drop Project to submit and validate your work



Drop Project

- Go to https://iadi.dropproject.org/ (you will be redirected to GitHub)
- Login using your GitHub account
- If you are authorized (need to fill the form github<->student first), you will see the link for the assignment
- The instructions for the assignment are accessible through that link
- All Drop Project submissions must have an AUTHORS.txt in the project root with this content (even if it is an individual submission):



Drop Project

- Two methods of submission:
  - Upload a zip file with the project
    (AUTHORS.txt, pom.xml, src folder)
  - Connect to a git repository

Drop Project

- Two methods of submission:
  - Upload a zip file with the project
    (AUTHORS.txt, pom.xml, src folder) ← The first “hello world” exercise
  - Connect to a git repository ← All other (real) exercises